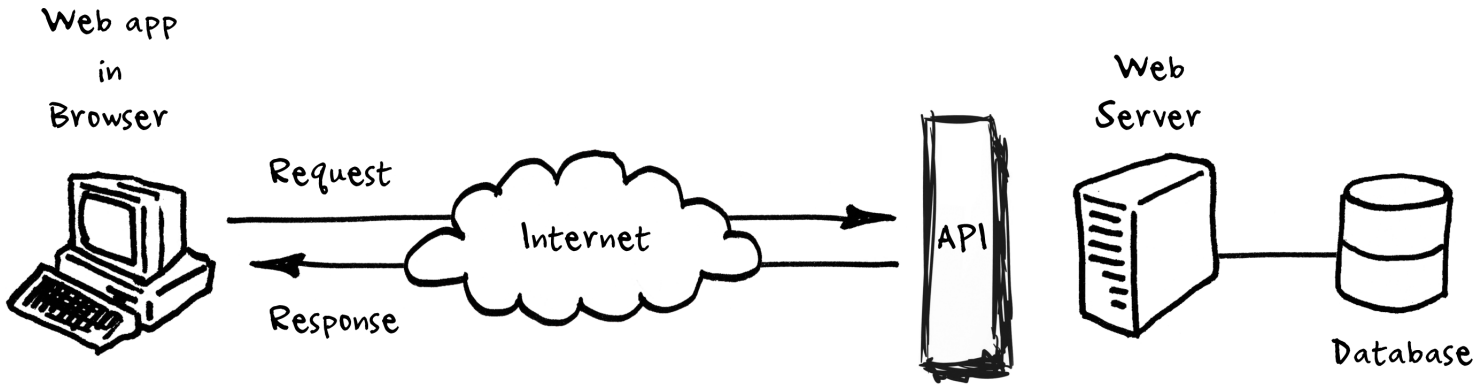
Web API using Java-RS

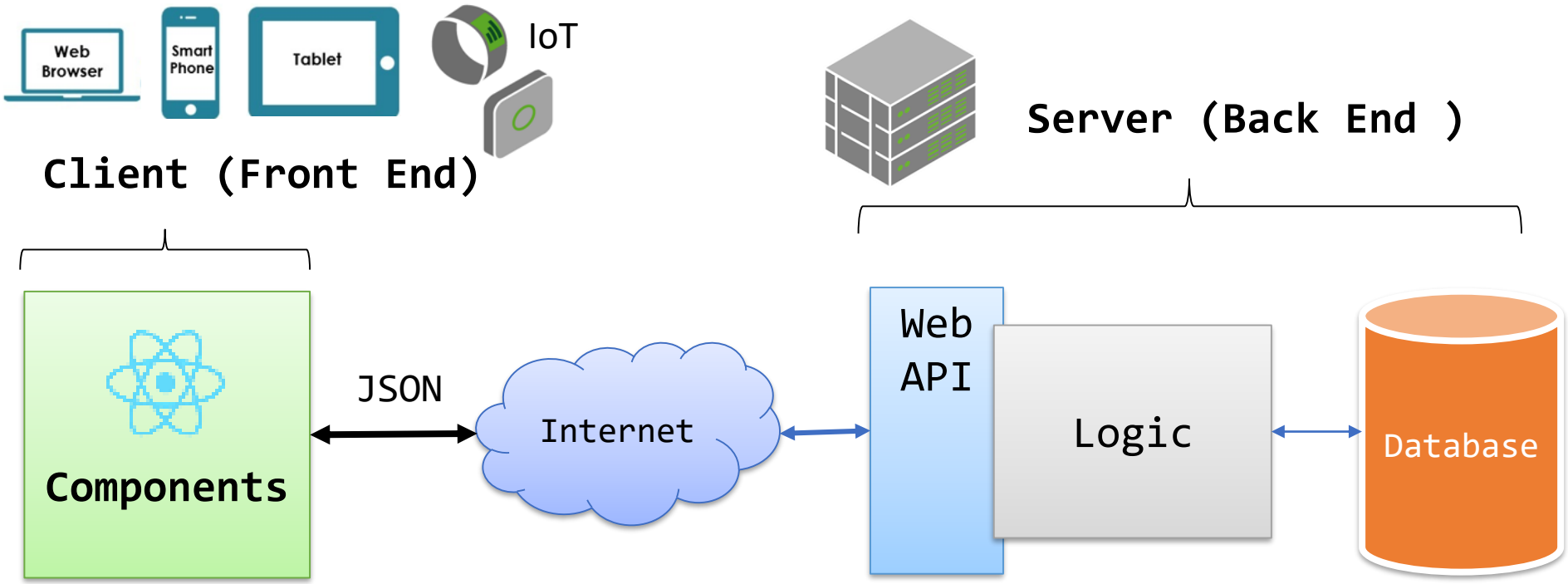


Outline

1. Web and HTTP
2. Web API
3. Web API Programming using JAX-RS

Components of Single Page Application (SPA) Architecture

- A **SPA** has **1 main shell page** and **multiple UI components loaded** in response to user actions
- Popular architecture of moderns Web/Mobile Apps



What is a Web API

- Web API: A set of methods exposed over the web via HTTP to allow **programmatic access to applications**
- Web API are designed for **broad reach**:
  - Can be accessed by a broad range of clients including browsers and mobile devices
  - Can be implemented or consumed in any language
- Uses HTTP as an **application protocol**


WWW

Web and HTTP

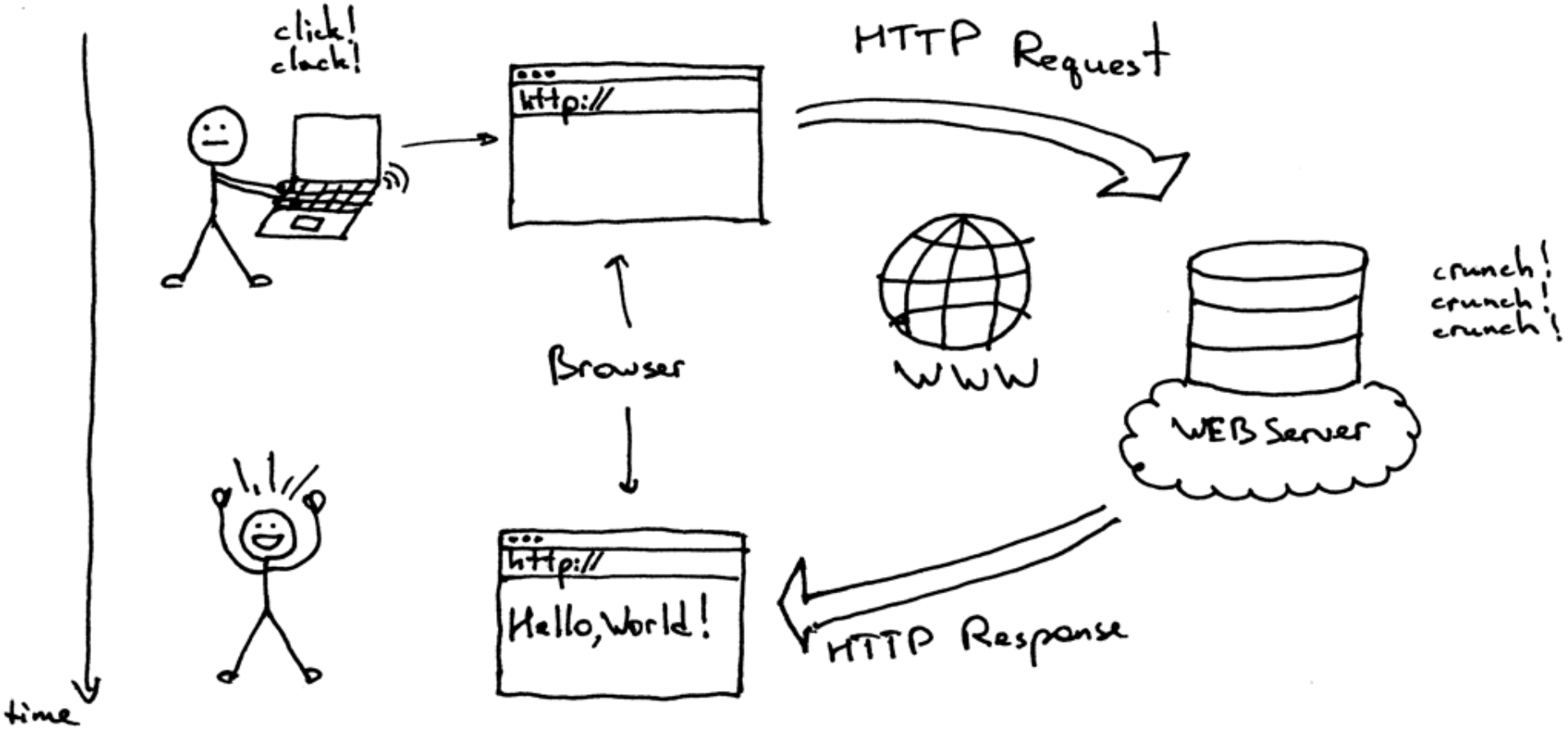

GET /index.html
HTTP/1.1
HTTP/1.1 200 OK
"Welcome to our
Web site!"

What is Web?

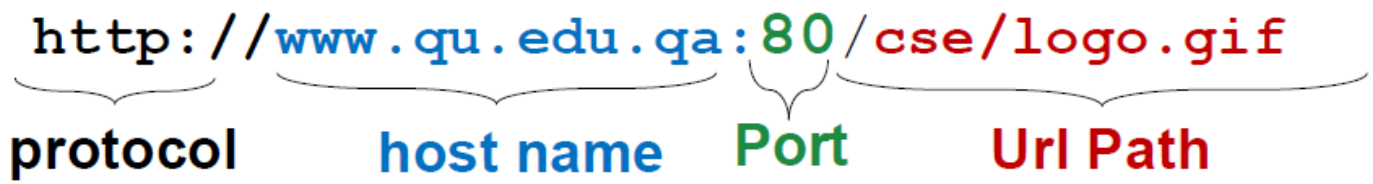
- Web = **global distributed system of interlinked resources accessed over the Internet using the HTTP protocol**
  - Consists of set of **resources** located on different servers:

    HTML pages, images, videos and other resources
  - Resources have unique **URL** (Uniform Resource Locator) address
  - Accessed through standard **HTTP** protocol
- The Web has a Client/Server architecture:
  - **Web browser** (client) requests resources (using HTTP protocol) and displays them
  - **Web server** sends resources in response to requests (using HTTP protocol)

How the Web Works?



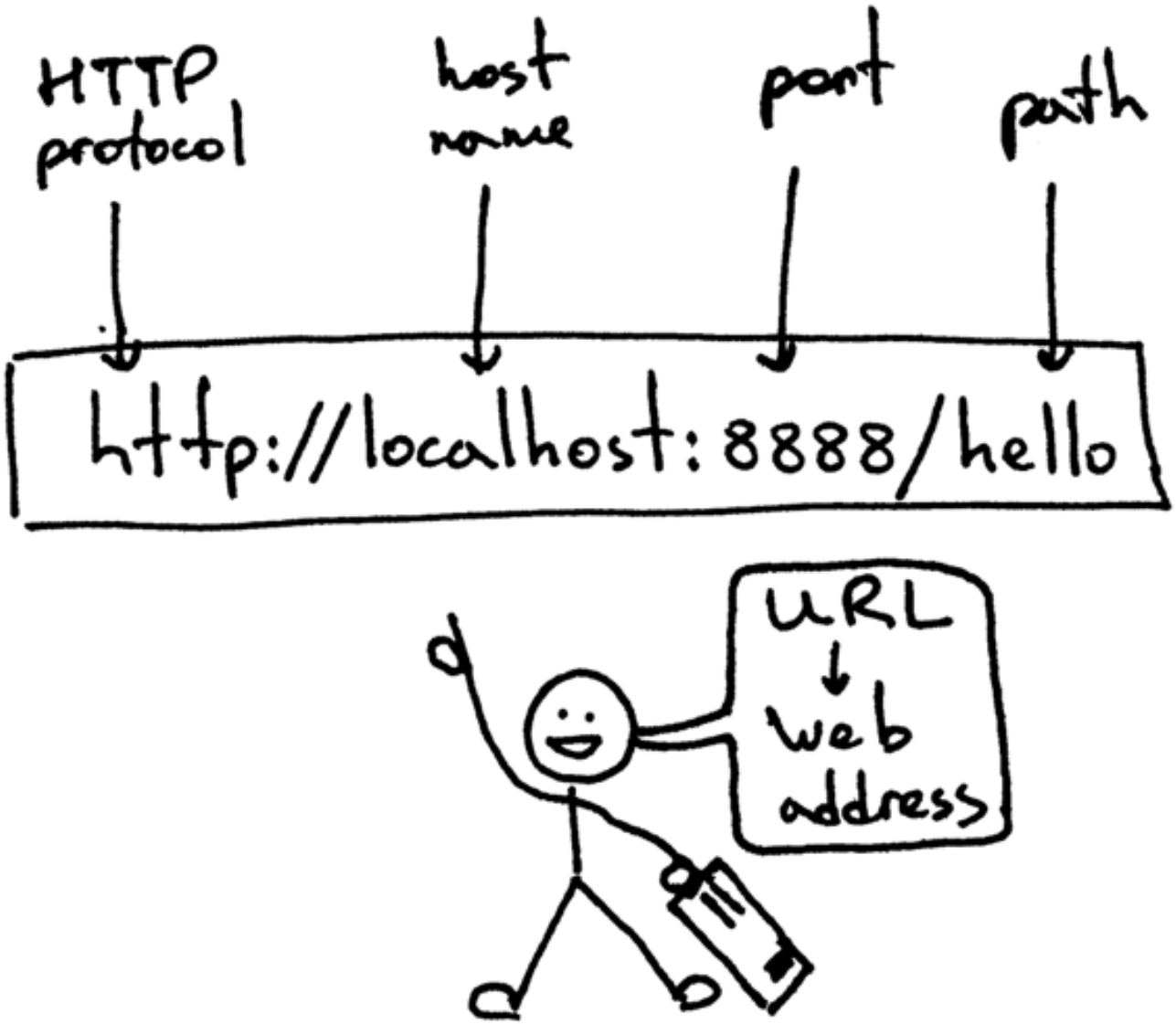
Uniform Resource Locator (URL)



- URL is a formatted string, consisting of:
  - **Protocol** for communicating with the server (e.g., http, https, ...)
  - **Name of the server or IP** address plus port (e.g. qu.edu.qa:80, localhost:8080)
  - **Path of a resource** (e.g. **/ceng/index.html**)
  - **Parameters** aka **Query String** (optional), e.g.

    https://www.google.com/search**?q=qatar%20university**

URL Example


HTTP protocol
host name
port
path
http://localhost:8888/hello
URL
web address

URL Encoding

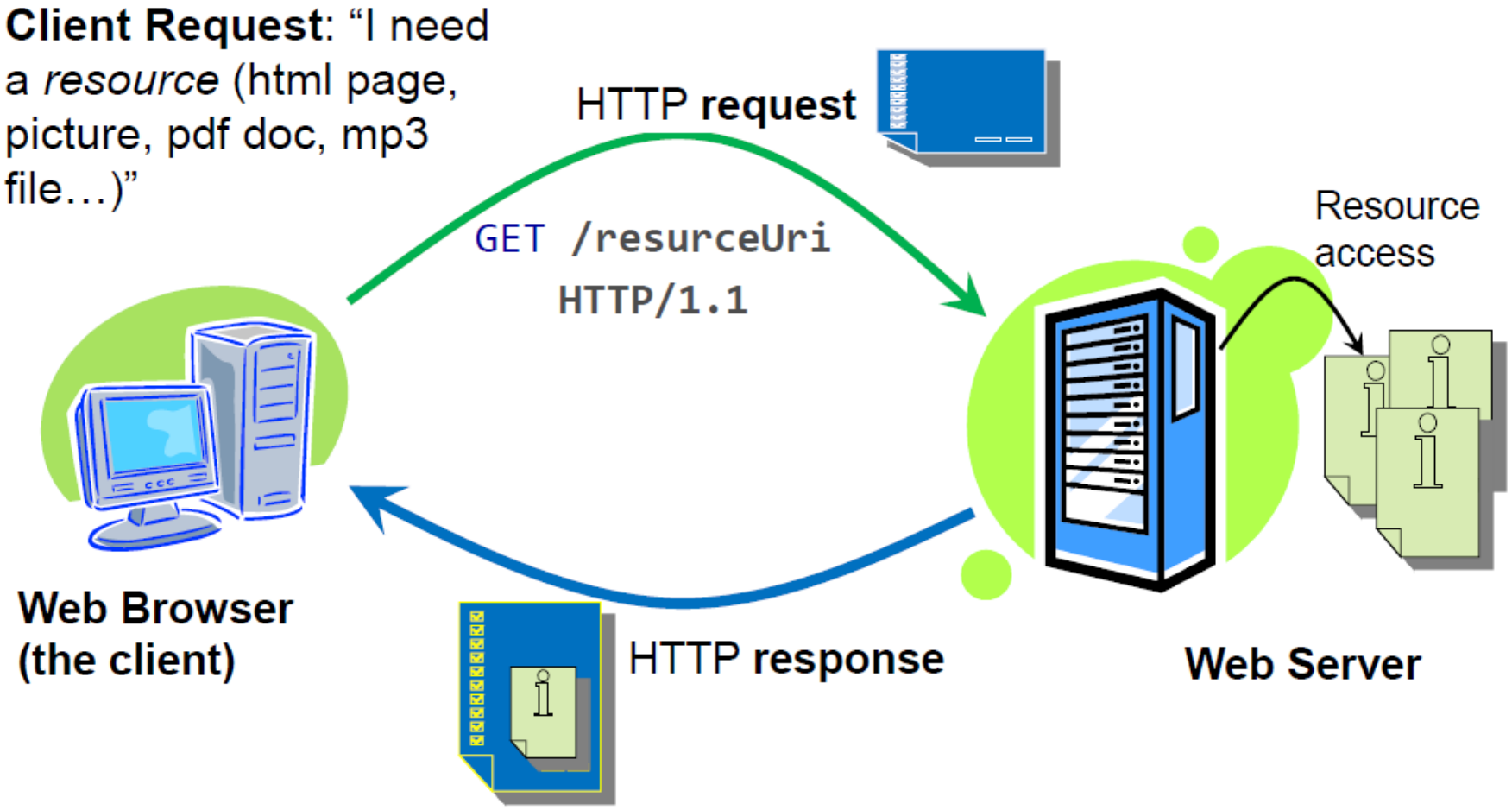
- According RFC 1738, the characters allowed in URL are alphanumeric [0-9a-zA-Z] and the special characters $-_.+!*'()
- Unsafe characters should be encoded, e.g.,

  http://google.com/search?q=qatar%20university

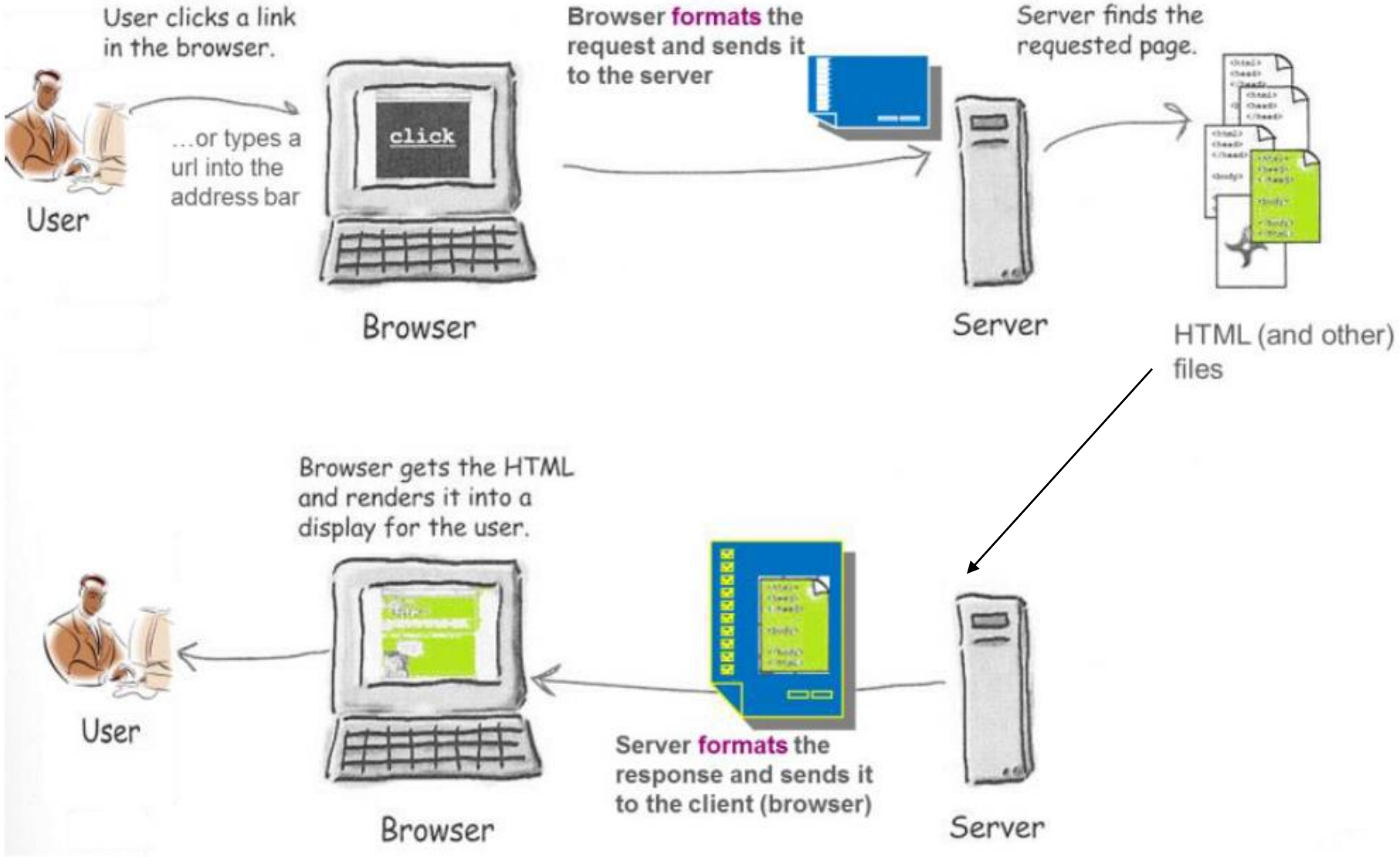
Commonly encoded values:

ASCII Character	URL-encoding
space	%20
!	%21
"	%22
#	%23
$	%24
%	%25
&	%26

Web uses Request/Response interaction model
HTTP is the *message protocol* of the Web

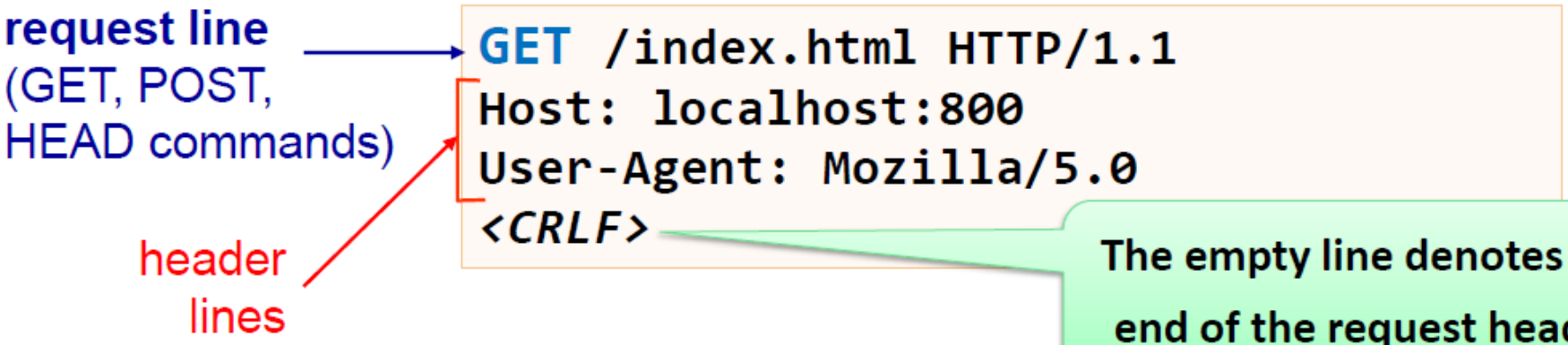


The sequence for retrieving a resource



Request and Response Examples

- **HTTP request:**



- **HTTP response:**

HTTP Request Message

- Request message sent by a client consists of

– **Request line** – request method (GET, POST, HEAD, ...), resource URI, and protocol version

– **Request headers** – additional parameters

– **Body** – optional data

  - e.g. posted form data, files, etc.

```
<request method> <URI> <HTTP version>
<headers>
<empty line>
<body>
```

Must-know Headers

- Content-Type
  - The resource representation form
    - E.g. application/xml, application/json
- Accept
  - Client tells server what formats it wants

HTTP Request Methods

- **GET**
  - – **Retrieve a resource** (could be static resource such as an image or a dynamically generated resource)
  - – Input is appended to the request URL E.g.,

  http://google.com/?q=Qatar
- **POST**
  - – **Create or Update a resource**
  - – Web pages often include form input. Input is submitted to server in the **message body.** E.g.,


```
POST /calc HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
num1=20&operation=*&num2=10
```

HTTP Response Message

- Response message sent by the server

– **Status line** – protocol version, status code, status phrase

– **Response headers** – provide metadata such as the Content-Type

– **Body** – the contents of the response (i.e., the requested resource)

```
<HTTP version> <status code> <status text>
<headers>
<empty line>
<response body>
```

HTTP Response – Example

status line (protocol status code status text)

Try it out and see HTTP in action using **HttpFox**

```
HTTP/1.1 200 OK
Content-Type: text/html
Server: QU Web Server
Content-Length: 131
<CRLF>
<html>
  <head><title>Calculator</title></head>
  <body>20 * 10 = 200
      <br><br>
      <a href='/calc'>Calculator</a>
  </body>
</html>
```

HTTP response headers

The empty line denotes the end of the response header

Response body. e.g., requested HTML file

Common Internet Media Types

- The **Content-Type** header describes the media type contained in the body of HTTP message
- **Full list @** http://en.wikipedia.org/wiki/MIME_type
- Commonly used media types (**type/**subtype):

Type/Subtype	Description
application/json	JSON data
image/gif	GIF image
image/png	PNG image
video/mp4	MP4 video
text/xml	XML
text/html	HTML
text/plain	Just text

HTTP Response Status Codes

- Status code appears in 1st line in the response message
- HTTP response code classes
  - 2xx: success (e.g., “200 OK”)
  - 3xx: redirection (e.g., "302 Found")

  "302 Found" is used for redirecting the Web browser to another URL
  - 4xx: client error (e.g., “404 Not Found”)
  - 5xx: server error (e.g., “503 Service Unavailable”)

Popular Status Codes

Code	Reason	Description
200	OK	Success!
301	Moved Permanently	Resource moved, don't check here again
302	Moved Temporarily	Resource moved, but check here again
304	Not Modified	Resource hasn't changed since last retrieval
400	Bad Request	Bad syntax?
401	Unauthorized	Client might need to authenticate
403	Forbidden	Refused access
404	Not found	Resource doesn't exist
500	Internal Server Error	Something went wrong during processing
503	Service Unavailable	Server will not service the request

Browser Redirection

- HTTP browser redirection example
  - HTTP GET requesting a moved URL:

(Request-Line)	GET /qu HTTP/1.1
Host	localhost:800
User-Agent	Mozilla/5.0 (Windows NT 6.3; WOW64; rv:27.0) Gecko/20100101 Firefox/27.0
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

  - The HTTP response says that the browser should request another URL:

(Status-Line)	HTTP/1.1 301 Moved Permanently
Location	http://qu.edu.qa

2

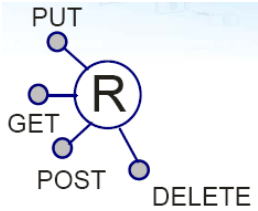
Web API
(aka REST Services)

What is a REST Service?

- Web API = Web accessible Application Programming Interface. Also known as REST Services.
- Web API is a web service that accepts requests and returns **structured data** (JSON in most cases)
  - Programmatically accessible at a particular URL
  - You can think of it as a Web page returning JSON instead of HTML

- Major goal = **interoperability between heterogeneous systems**

REST Principles



- **Resources have unique address** (**nouns**) i.e., a URI

e.g., http://example.com/customers/123

- **Can use a Uniform Interface** (**verbs**) to access them:

  - HTTP verbs: GET, POST, PUT, and DELETE

- **Resource has representation(s)** (**data format**)

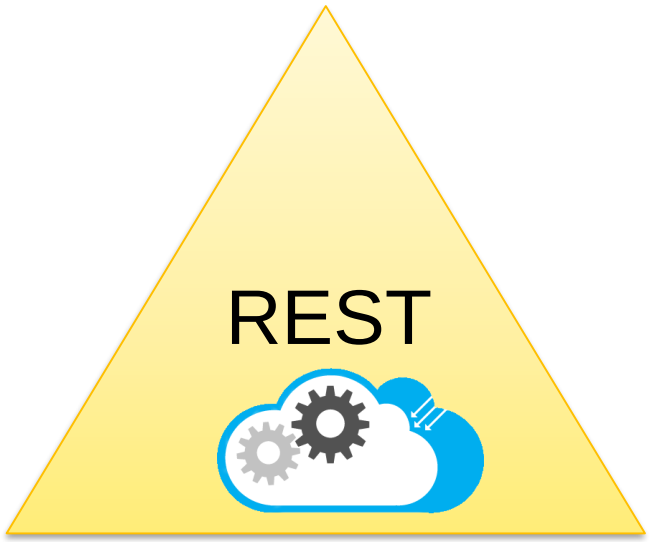
  - A resource can be in a variety of data formats: JSON, XML, RSS..

Resources

- The key abstraction in REST is a **resource**
- A resource is a conceptual mapping to a set of entities
  - Any **information that can be named can be a resource**: a document or image, a temporal service (e.g. "today's weather in Doha"), a collection of books and their authors, and so on

REST Services Main Concepts

Nouns (Resources)
e.g., http://example.com/employees/12345



Verbs
e.g., GET, POST

Representations
e.g., XML, JSON

Naming Resources

- REST uses URL to identify resources

> Dedicated **api** path is recommended for better organization

  - http://localhost/**api**/books/
  - http://localhost/api/books/ISBN-0011
  - http://localhost/api/books/ISBN-0011/authors

  - http://localhost/api/classes
  - http://localhost/api/classes/cmps356
  - http://localhost/api/classes/cs356/students

- As you traverse the **path** from more generic to more specific, you are navigating the data

Web API URI Design

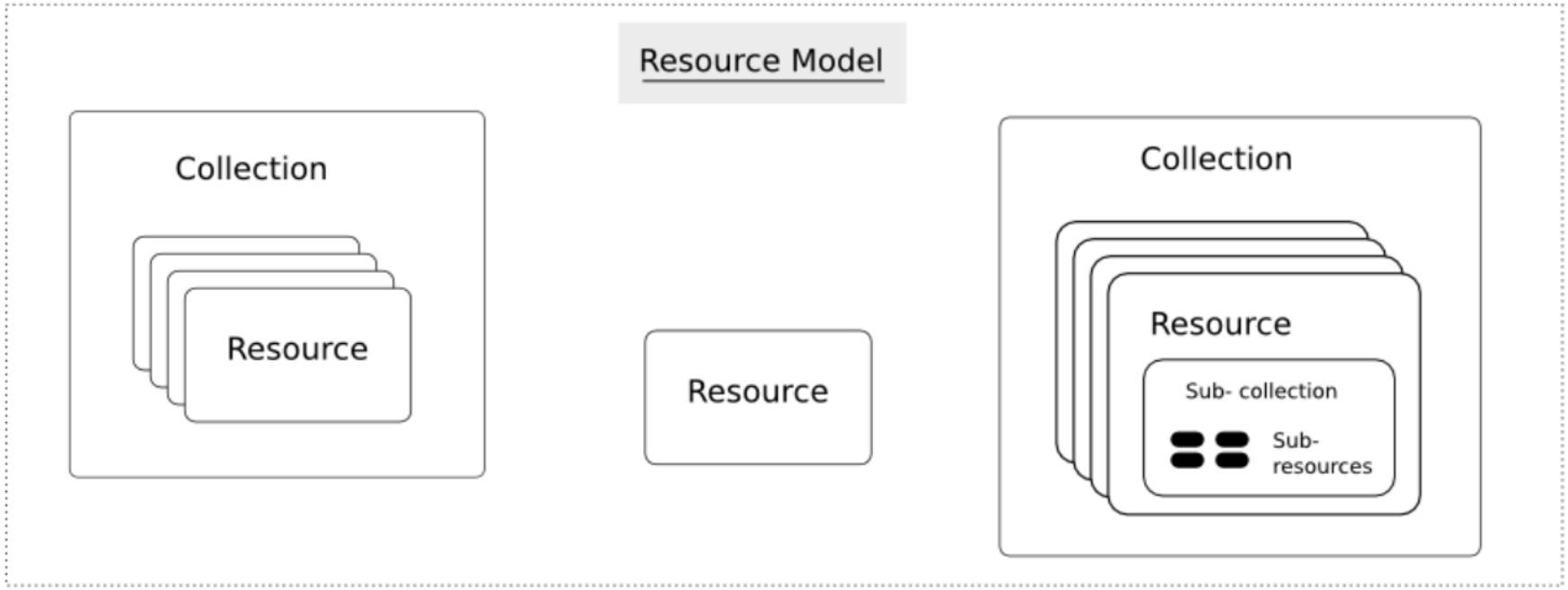
- Web API endpoints are *resources* and identified with a URI
- For examples in an order management application we might have 3 top-level resources

```
/orders
/orders/{id}
/products
/products/{id}
/customers
/customers/{id}
```

Example CRUD (Create, Read, Update and Delete) API that manages books

- Create a new book
  - **POST** /books
- Retrieve all books
  - **GET** /books
- Retrieve a particular book
  - **GET** /books/:id
- Replace a book
  - **PUT** /books/:id
- Update a book
  - **PATCH** /books/:id
- Delete a book
  - **DELETE** /books/:id

A Collection with Resources



A Collection with Resources

A Singleton Resource

Sub-collections and Sub-resources

Representations

Two main formats:

- **JSON**

```
{
    code: 'cmp123',
    name: 'Web Development'
}
```

- **XML**

```
<course>
    <code>cmp123</code>
    <name>Web Development</name>
</course>
```

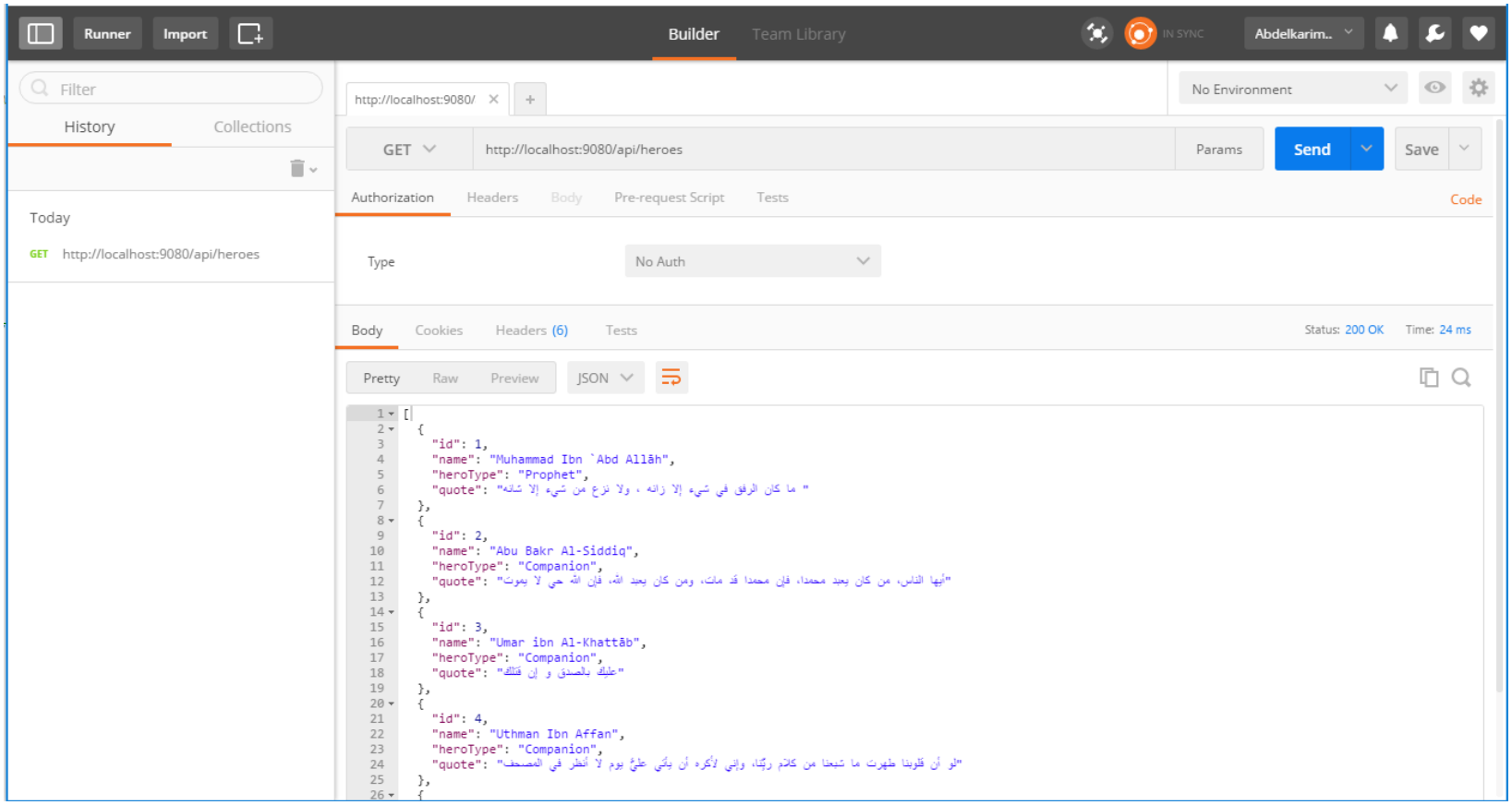
HTTP Verbs

- Represent the actions to be performed on resources
- Retrieve a representation of a resource: **GET**
- Create a new resource:
  - Use **POST** when the server decides the new resource URI
    - Post is not repeatable
  - Use **PUT** when the client decides the new resource URI
    - Put is repeatable
- **PUT** is typically used for update
- Delete an existing resource: **DELETE**
- Get metadata about an existing resource: **HEAD**
- Get which of the verbs the resource understands: **OPTIONS**

Testing REST Services

- Using Postman to test Web API

https://www.getpostman.com/postman



Web API Programming using JAX-RS

Web API in Java: JAX-RS

- JAX-RS is Java API for creating Web API
- JAX-RS uses annotations to simplify the development of Web API and clients
- Many implementations are available:
  - **Jersey**: reference implementation from Oracle
  - Apache CXF
  - RESTEasy, JBoss implementation.

@Path

- Specifies the URI Path of the resource corresponding to a class

```
@Path("/contacts")
@Stateless
public class ContactService {
```

- Query strings / parameters can be embedded in the URI, and then retrieved with the **@PathParam** annotation

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getContact(@PathParam("id") int contactId) {
    Contact contact = contactRepository.getContact(contactId);
    if (contact != null) {
```

Sub-resources

- **@Path** may be used on classes and such classes are referred to as root resource classes
- **@Path** may also be used on methods of root resource classes

```
@Path("/contacts")
@Stateless
public class ContactService {
    @GET
    @Path("/countries")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getCountries() {
        List<String> countries = contactRepository.getCountries();
        String json = (new Gson()).toJson(countries);
        System.out.println(json);
        return Response.ok(json).build();
    }
```

HTTP Methods

- **@GET**, **@PUT**, **@POST**, **@DELETE** annotations correspond to the HTTP methods
  - Represent the actions to be performed on resources

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addContact(Contact contact) throws URISyntaxException {
    contact = contactRepository.addContact(contact);
    String location = String.format("/contacts/%s", contact.getContactId());
    String msg = String.format("contact #%s created successfully", contact.getCo
    return Response.created(new URI(location)).entity(msg).build();
}

@PUT
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response updateContact(Contact contact) {
    contactRepository.updateContact(contact);
    String msg = String.format("Contact #%s updated sucessfully", contact.getCo
    return Response.ok(msg).build();
}
```

@Produces

- Specifies the MIME media types of **representations** a resource can produce typically XML or JSON
  - Multiple representations of the same resource are allowed

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getContact(@PathParam("id") int contactId) {
    Contact contact = contactRepository.getContact(contactId);
    if (contact != null) {
        return Response.ok(contact).build();
    } else {
        String msg = String.format("Contact # %d not found", contactId);
        return Response.status(Response.Status.NOT_FOUND).entity(msg).build();
    }
}
```

@Consumes

- It is used to specify the MIME media types of representations a resource can consume

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addContact(Contact contact) throws URISyntaxException {
    contact = contactRepository.addContact(contact);
    String location = String.format("/contacts/%s", contact.getContactId());
    String msg = String.format("contact #%s created successfully", contact.getCo
    return Response.created(new URI(location)).entity(msg).build();
}
```

Building Responses

- **Response** class can be used to build the HTTP response to return to the caller
  - You may specify the response headers such as the **status code** and the **location** of a newly created resource

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addContact(Contact contact) throws URISyntaxException {
    contact = contactRepository.addContact(contact);
    String location = String.format("/contacts/%s", contact.getContactId());
    String msg = String.format("contact #%s created successfully", contact.getC
    return Response.created(new URI(location)).entity(msg).build();
}
```

Contextual Dependency Injection (CDI Bean)

Class + Annotation = Bean

- A Java Bean is an annotated class that has a no-argument constructor.
- Bean specifies what it needs through **annotations**
- **Container provides requested services** to the bean

=> Allows the programmer to write less and focus on delivering clean business logic

Contextual Dependency Injection (CDI Bean)

- CDI Bean is annotated class that can be injected
- Should the bean exist only during the lifespan of a request? If so, use **`@RequestScoped`**

- Should the bean exist during the lifespan of the application and should the state of the bean be shared between all the requests? If so, use **`@ApplicationScoped`**

How the client access an EJB?

- Use **@Inject <Java-Type> <variable>**
- **<Java-Type>** can be Java class or Java interface
- Bean can be injected at “Injection points”
  - Class level variable or Method parameter

```
public class MyGreeter {

   // Inject Greeting object for field injection
   @Inject Greeting greeting;

   public sayGreeting(String name){
      // You can then used the injected Greeting object
      System.out.println(greeting.greet(name));
   }
}
```

The **`@Inject`** annotation marks ***greeting*** as a dependency injected attribute

The **Dependency Injector** automatically injects an instance of **Greeting** at runtime

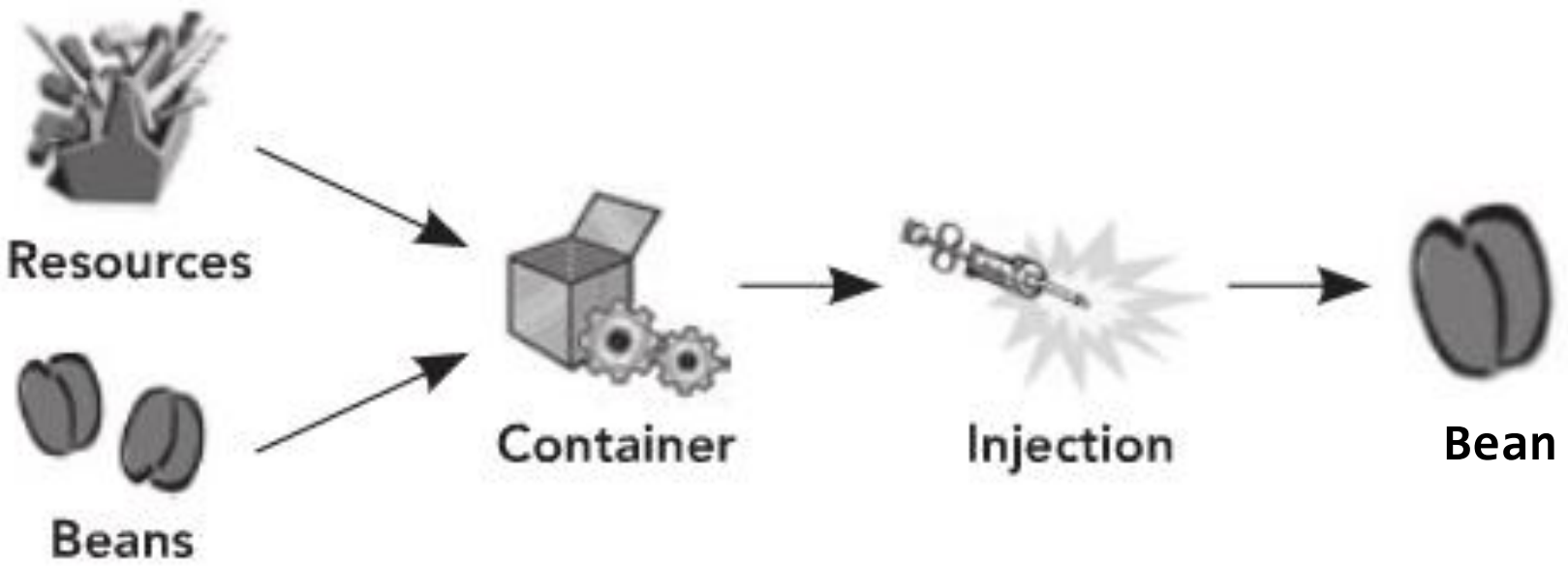
What is and Why Dependency Injection?

- Classes specify what their dependencies are NOT how to obtain them
  - It is now the container's job to create/find objects
- Why Dependency injection?
  - Allows the container to do the bean-discovery and bean-wiring

  ⇒ Easier to switch from one implementation to another
  - Loose Coupling – lower dependency between the server class and the clients using it.

  ⇒ This reduces cost of change & ease unit testing

Dependency injection



Dependency injection = declare component dependencies and let the ***Dependency Injector*** deal with the complexities of instantiating, initializing, and supplying object or resource references to clients as required.

Hosting Java Web API

- **Helidon** is the new lightweight Java framework that has been open sourced recently by Oracle to build microservices-based applications

https://helidon.io/

- Helidon microservice is a **Java SE application** that starts a tinny HTTP server from the main method.
  - An application server is not required.
  - App uses a tiny Web server powered by Netty https://netty.io

Helidon Architecture