

SEW

IT-Medientechnik

NVS

Informatik

RESTful Web

Representational State Transfer

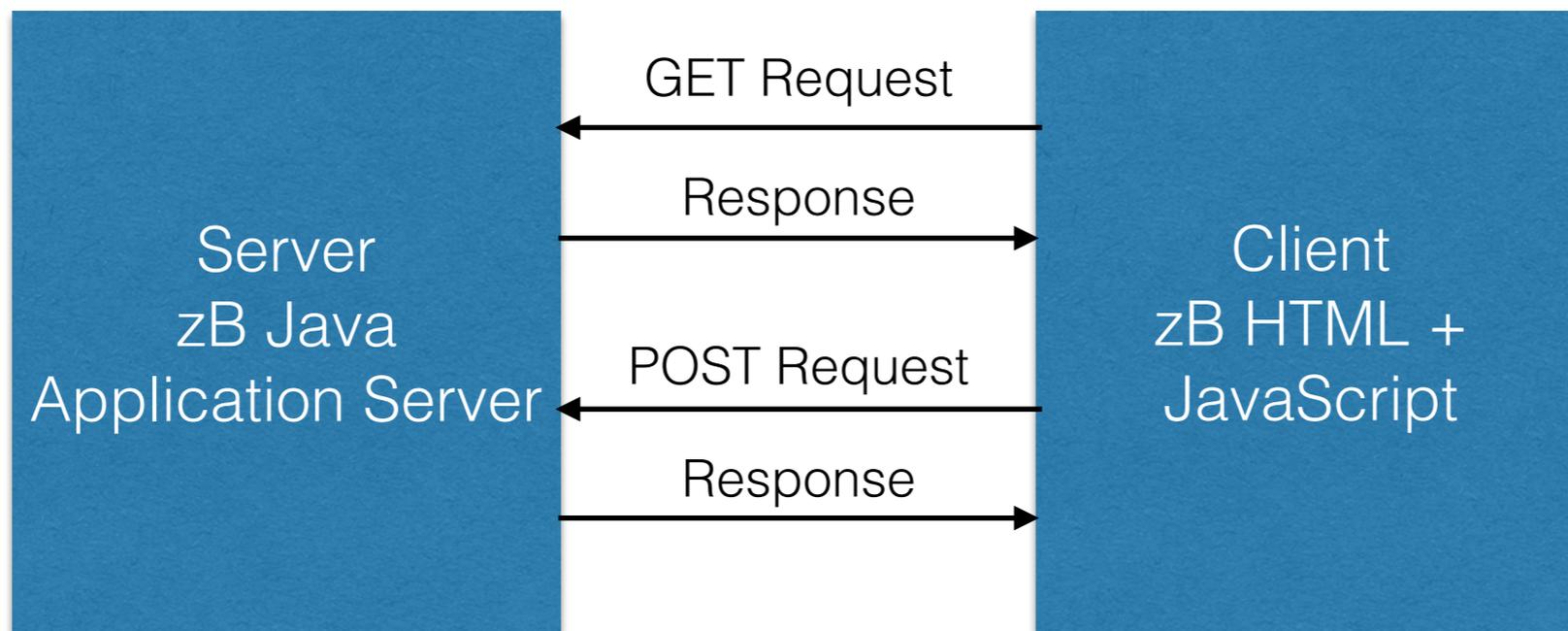
Warum REST?

- REST ist die Lingua Franca des Webs
- Heterogene (verschiedenartige) Systeme können mit REST kommunizieren, unabhängig von der Technologie der beteiligten Systeme
- REST Ressourcen werden im Web angeboten: zB Google Maps, Twitter, Youtube, Facebook

<http://www.programmableweb.com/apis/directory>

Funktionsweise

- Sämtliche Informationen im Web sind Ressourcen, auf die mittels HTTP zugegriffen werden kann.



Anforderungen

- **Adressierbarkeit:** Jede Ressource muss über einen eindeutigen Unique Resource Identifier (kurz URI) identifiziert werden können. Ein Kunde mit der Kundennummer 123456 könnte also zum Beispiel über die URI `http://localhost/customers/123456` adressiert werden.
- **Zustandslosigkeit:** Die Kommunikation der Teilnehmer untereinander ist zustandslos. Dies bedeutet, dass keine Benutzersitzungen (etwa in Form von Sessions und Cookies) existieren, sondern dass bei jeder Anfrage alle notwendigen Informationen wieder neu mitgeschickt werden müssen.
- **Einheitliche Schnittstelle:** Auf jede Ressource muss über einen einheitlichen Satz von Standardmethoden zugegriffen werden können. Beispiele für solche Methoden sind die Standard-HTTP-Methoden wie GET, POST, PUT, und mehr.
- **Entkopplung von Ressourcen und Repräsentation:** Das bedeutet, dass verschiedene Repräsentationen einer Ressource existieren können. Ein Client kann somit etwa eine Ressource explizit beispielsweise im XML- oder JSON-Format anfordern.

HTTP Methoden

GET

READ: Lesen einer bestimmten Ressource

POST

CREATE (und UPDATE): Erstellen oder Ändern ohne einer ID

PUT/PATCH

(CREATE und) UPDATE: Erstellen oder Ändern mit einer bekannten ID

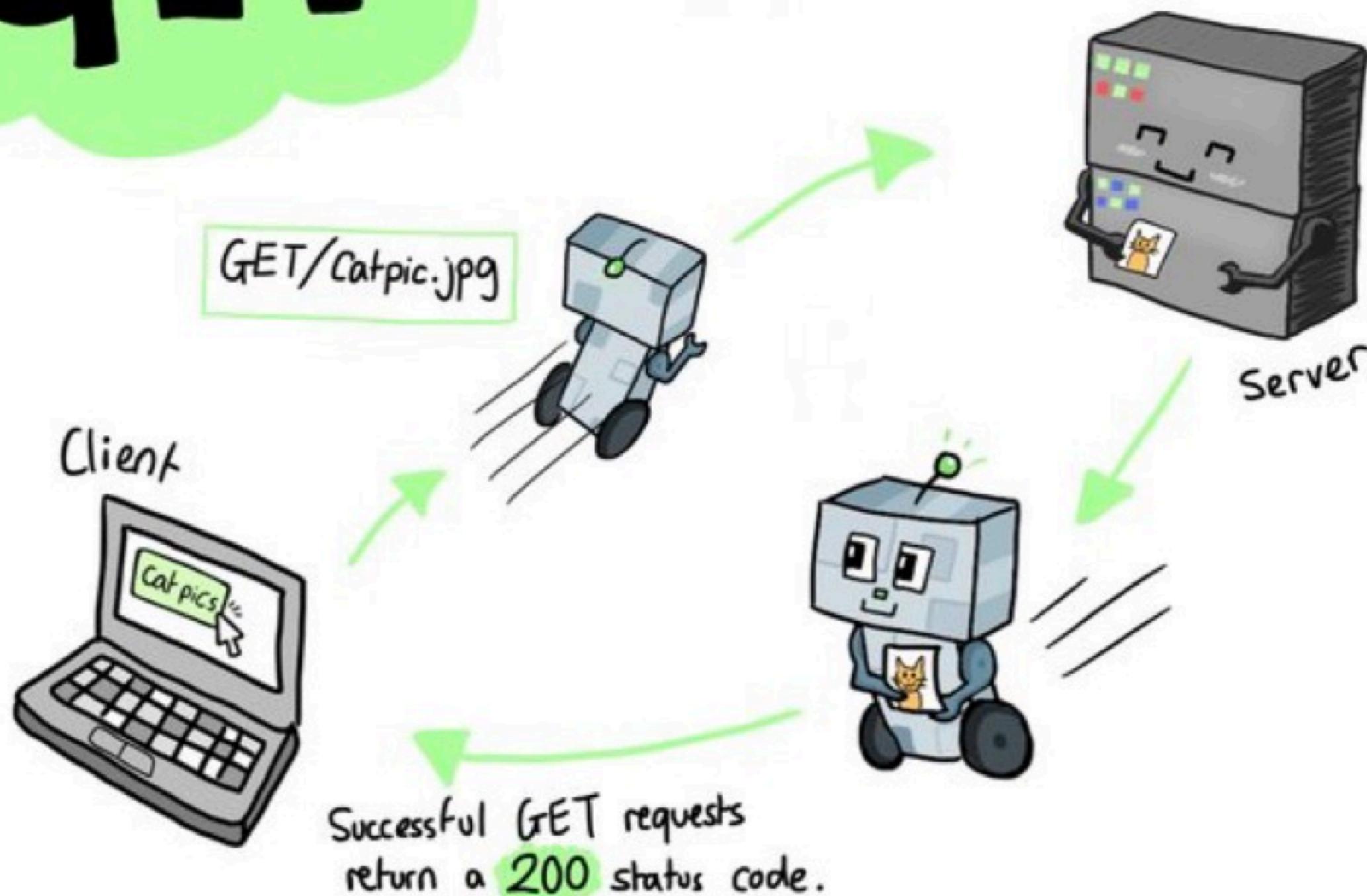
DELETE

DELETE: Löschen einer Ressource

GET

GET requests retrieve
a resource from a server.

GET requests are cacheable and idempotent (do not
affect the status of the server).



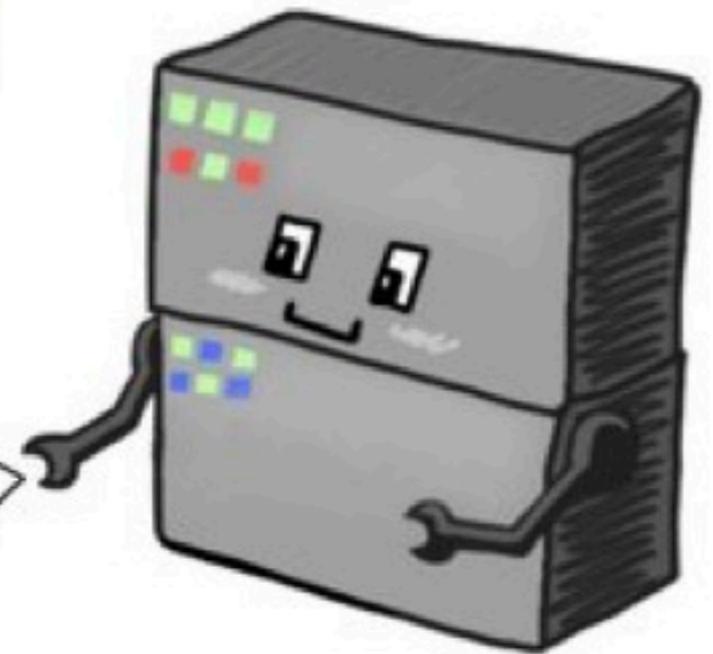
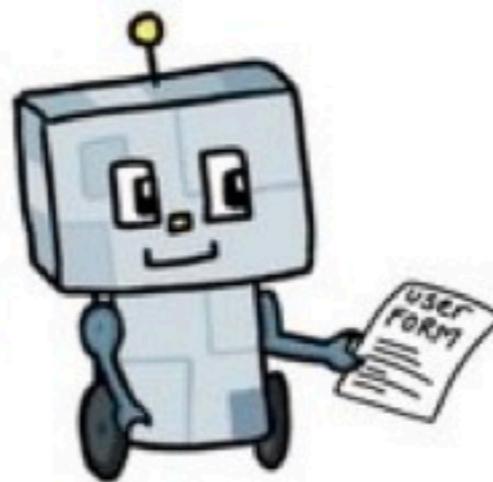
POST

POST requests submit information to the server.

They change the state of the server, so they are not idempotent. They are also not cacheable.



POST / user_form.php

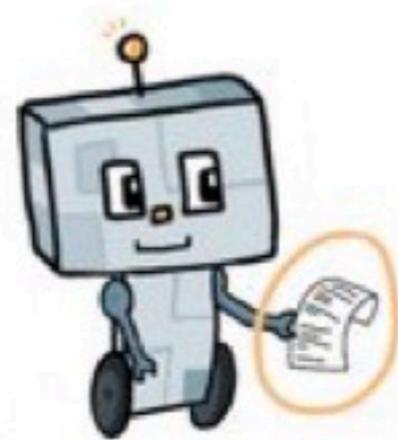


Successful POST requests will return a **201** status code (created).

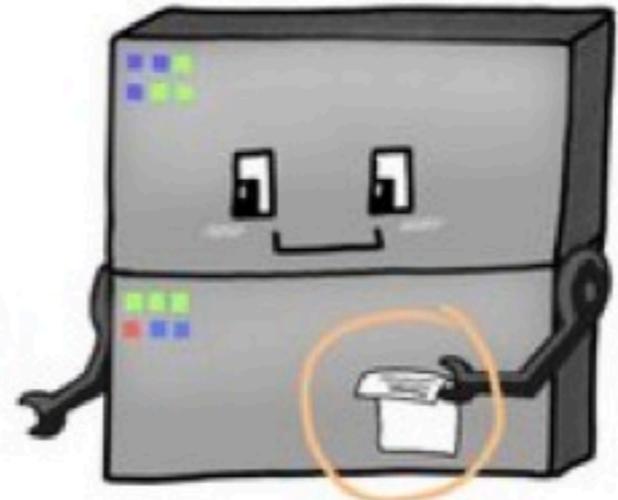
PUT

PUT is also used to create/update resources, but unlike POST, it is idempotent

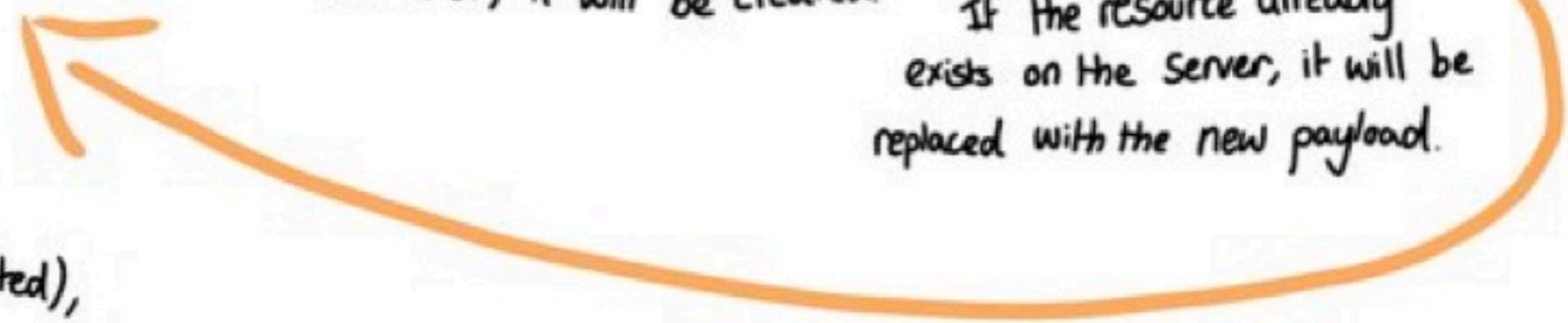
Calling multiple of the same PUT request does not affect the server.



If the resource does not exist, it will be created.



If the resource already exists on the server, it will be replaced with the new payload.



On Success, will return a **200** (OK), **201** (created), or **204** (no content) status.

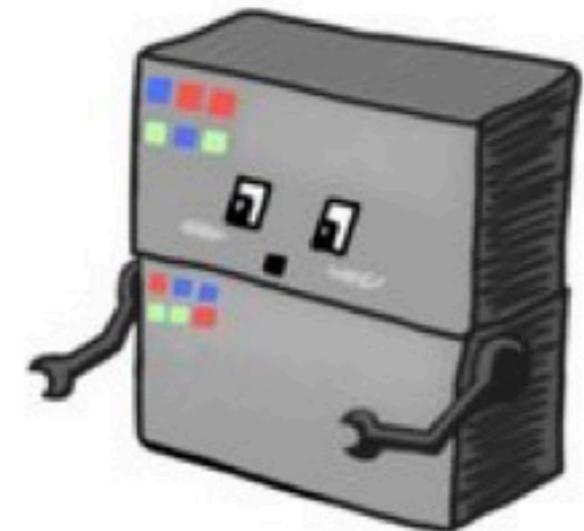
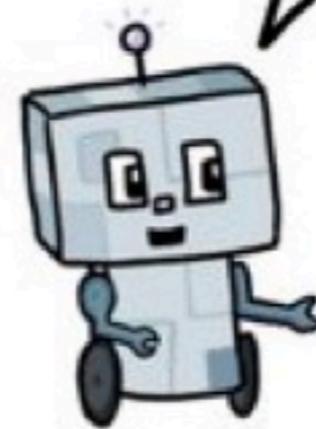
PATCH

Like PUT, PATCH also updates a resource, but PATCH only modifies a specified part of it instead of updating the entire resource.



```
PATCH /users/1  
{ "email": "coolbot96@robotmail.com" }
```

Coolbot96@robotmail.com ❌
↓
Coolbot97@robotmail.com ✅



Successful
PATCH methods
will return **2XX**

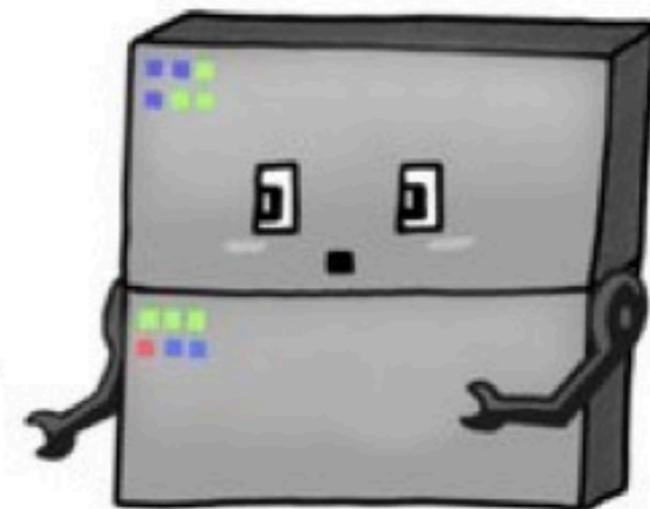
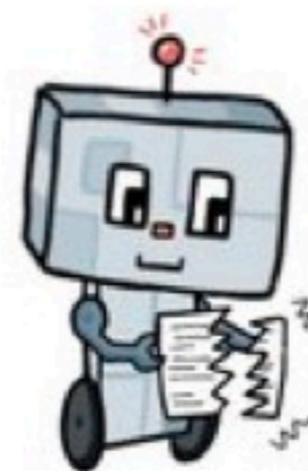
DELETE

The delete method is idempotent.

As suggested, it deletes a specified resource on the server.



DELETE /file.html



Successful DELETE requests return a 202 (accepted), 200 (OK), or 204 (no content) status.

DELETE requests the server to remove the resource identified by the request URL.

Beispiele

Request	Response
<pre>GET /products HTTP/1.0 Accept: application/json</pre>	<pre>HTTP/1.0 200 OK Content-Type: application/json Content-Length: 1234 [{ uri: "http://ws.mydomain.tld/products/1000", name: "Gartenstuhl", price: 24.99 }, { uri: "http://ws.mydomain.tld/products/1001", name: "Sonnenschirm", price: 49.99 }]</pre>

Beispiel POST

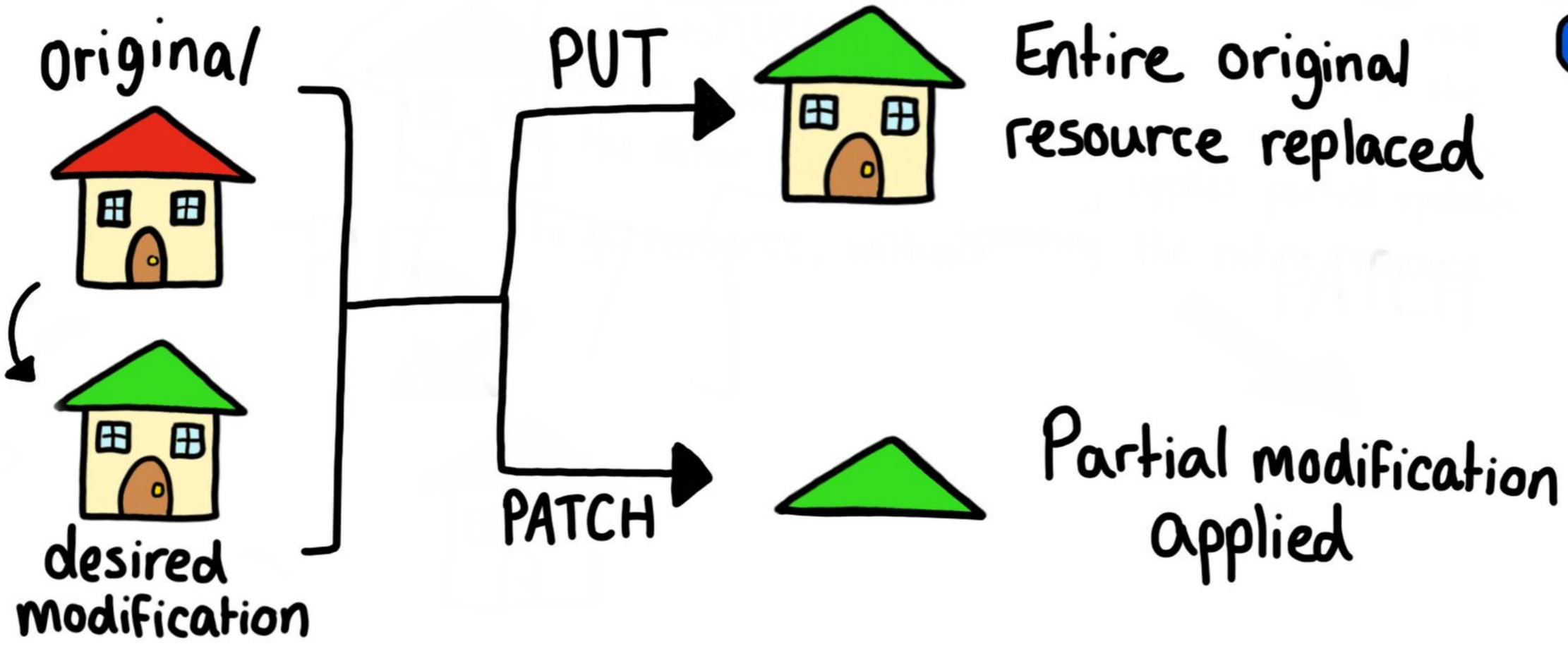
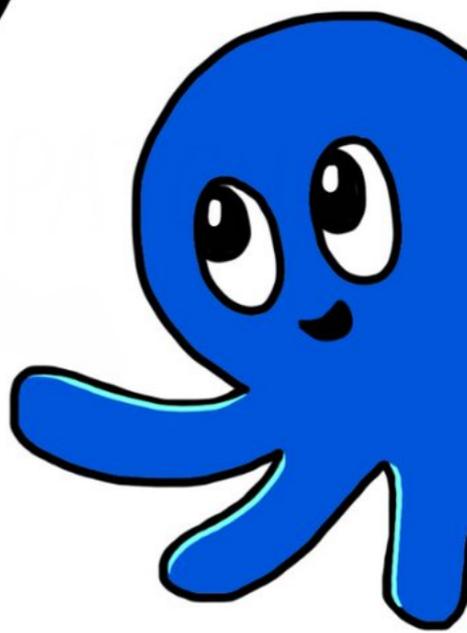
Request	Response
<pre>POST /products HTTP/1.0 Content-Type: application/json Content-Length: 38 { name: "Sandkasten", price: 89.99 }</pre>	<pre>HTTP/1.0 201 Created Location: http://ws.mydomain.tld/products/1002</pre>

Begriffe

- **Safety** (Sicherheit):
Daten werden nicht verändert
- **Idempotence**
(Idempotenz): Die Ressource behält auch bei mehrmaligen Aufruf den gleichen Zustand.

HTTP Method	Safe	Idempotent
GET	✓	✓
POST	✗	✗
PUT	✗	✓
DELETE	✗	✓
OPTIONS	✓	✓
HEAD	✓	✓

APIs use several HTTP methods to create, read, update, and delete data. Two of these are PUT and PATCH, which are both used to modify and update data. However they both do it in slightly different ways.



As a summary, we can say that PUT uses the request URI to deliver a modified version of the resource that replaces the original.

On the other hand, PATCH only applies partial updates without updating the entire resource.

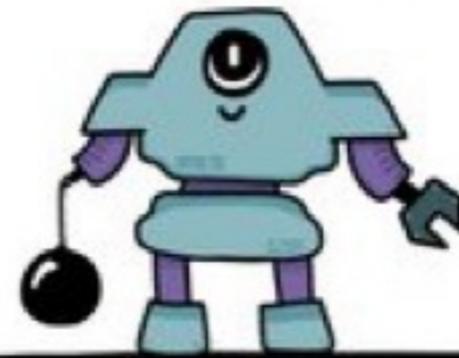


Here is a summary of the difference in features between PUT and PATCH. In general, it's best to use PATCH unless you need to update the whole resource. Using the PUT method when you only need to modify a few fields can use an unnecessary amount of bandwidth.

An HTTP Method is considered safe if it does not alter the state of the server.

If the effect of making one request is the same as making multiple identical requests, the method is idempotent.

Cacheable HTTP responses can be stored and retrieved for use at another time.



	PUT	PATCH
Safe	No	No
Idempotent	yes	Officially No, but it depends on the request's action
Cacheable	No	No

RESTful API

GET PUT POST DELETE

Ein erstes Beispiel

Jakarta EE

The image shows a sequence of five steps in the IntelliJ IDEA 'New Project' dialog:

- 1**: Selecting 'Maven' from the project SDK list.
- 2**: Entering the Maven coordinates: GroupId: `at.htl.restprimer`, ArtifactId: `restprimer`, and Version: `1.0-SNAPSHOT`.
- 3**: Entering the project name 'RestPrimer' and the project location `/Users/stuetz/Dropbox/htl/skripten/java.4jg.nvs/presentations/04.RestPrimer/neu/RestPrimer`.
- 4**: Clicking the 'Finish' button to complete the project creation.
- 5**: A message box indicating that 'Maven plugins need to be imported' and providing links for 'Import Changes' and 'Enable Auto-Import'.

pom.xml



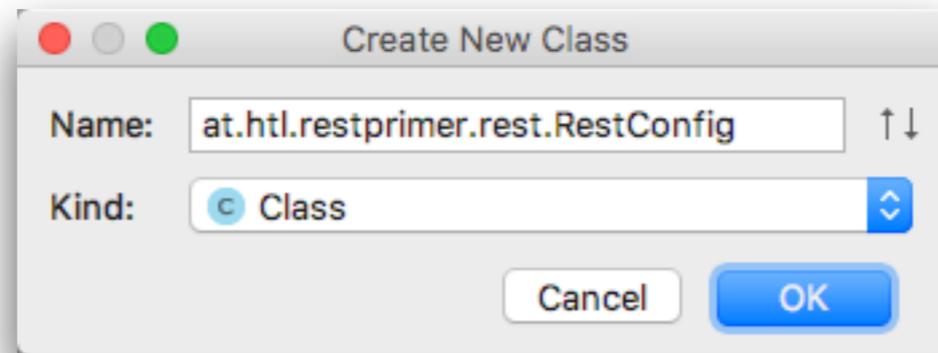
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <packaging>war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

  <groupId>at.htl</groupId>
  <artifactId>restprimer</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>jakarta.platform</groupId>
      <artifactId>jakarta.jakartaee-api</artifactId>
      <version>8.0.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <build>
    <finalName>restprimer</finalName>
  </build>
</project>
```

REST-Konfiguration



```
package at.htl.restprimer.rest;  
  
public class RestConfig extends App {  
}
```

- @ ApplicationPath (javax.ws.rs)
- Application (javafx.application)
- Application (javax.faces.application)
- Application (javax.ws.rs.core)
- Application (com.apple.eawt)
- Application (com.sun.glass.ui)
- Application (com.sun.javafx.tools.ant)

BEACHTEN: Wir verwenden javax.ws.rs !!!

```
package at.htl.restprimer.rest;  
  
import javax.ws.rs.ApplicationPath;  
import javax.ws.rs.core.Application;  
  
@ApplicationPath("api")  
public class RestConfig extends Application {  
}
```

Das ist die komplette Konfiguration, um REST-Ressourcen verwenden zu können. Ev. könnte man noch CORS konfigurieren. Warum wohl?

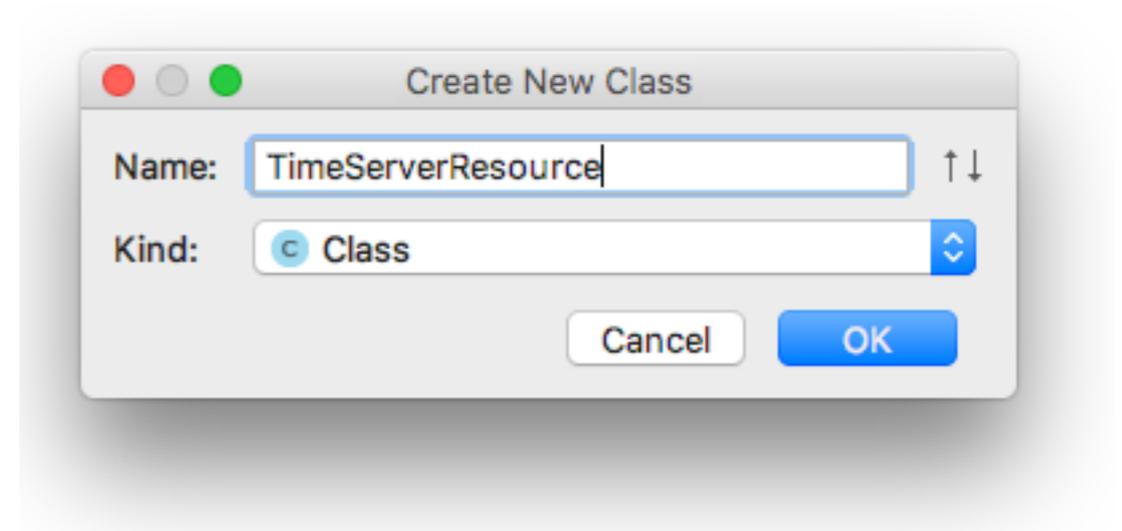
REST Ressource

```
package at.htl.restprimer.rest;

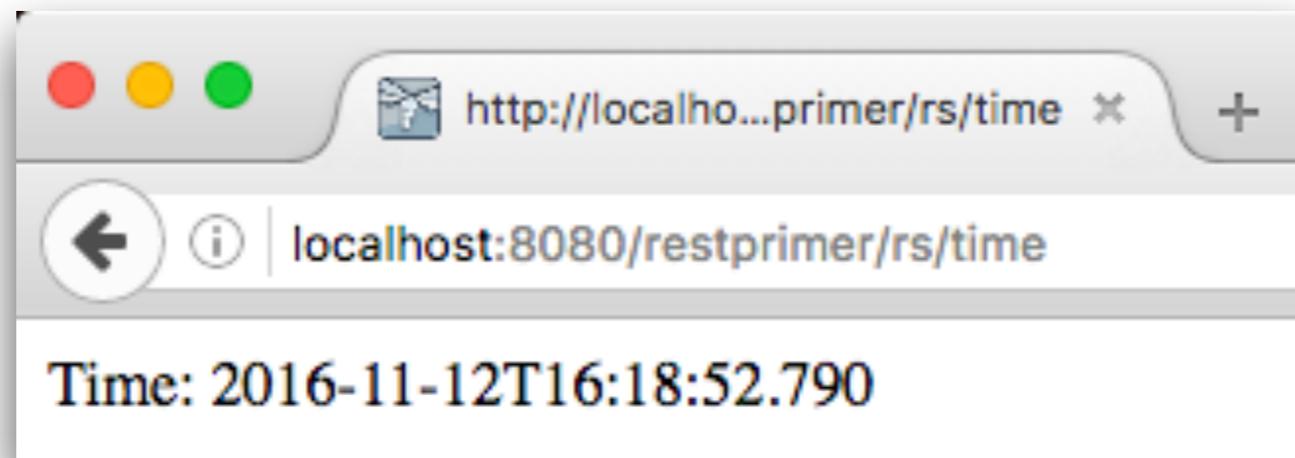
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import java.time.LocalDateTime;

@Path("time")
public class TimeServerResource {

    @GET
    public String time() {
        return "Time: " + LocalDateTime.now();
    }
}
```



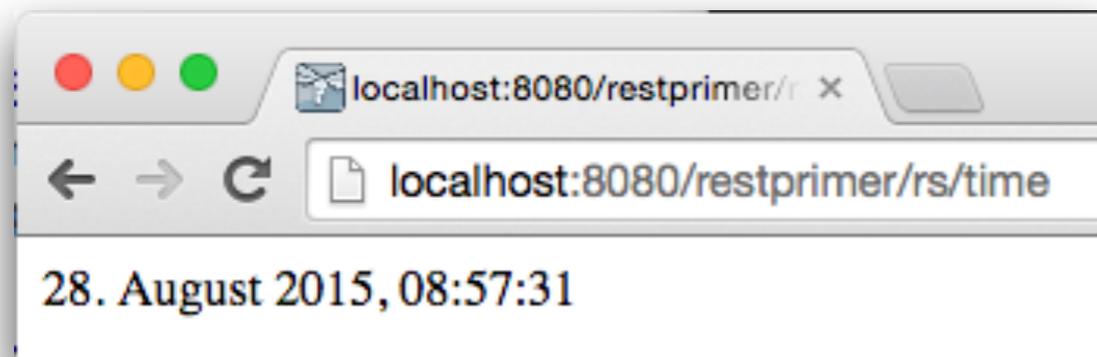
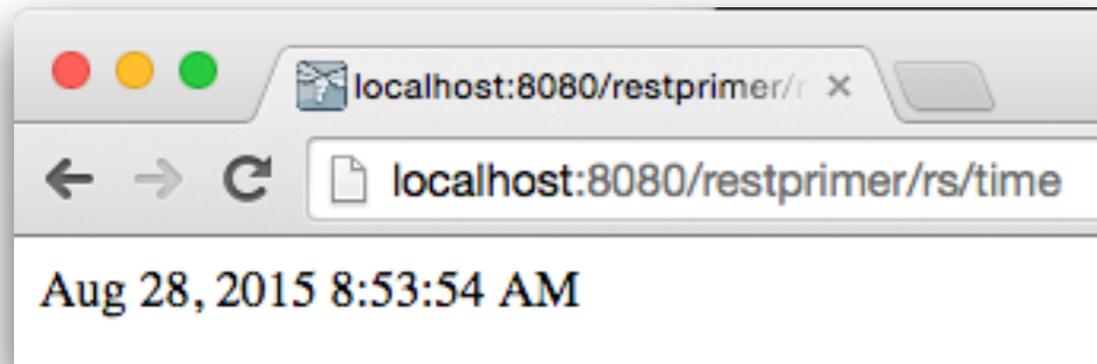
Testlauf



Nicht vergessen: Vor dem ersten Start den Application Server („Run/Debug“) konfigurieren

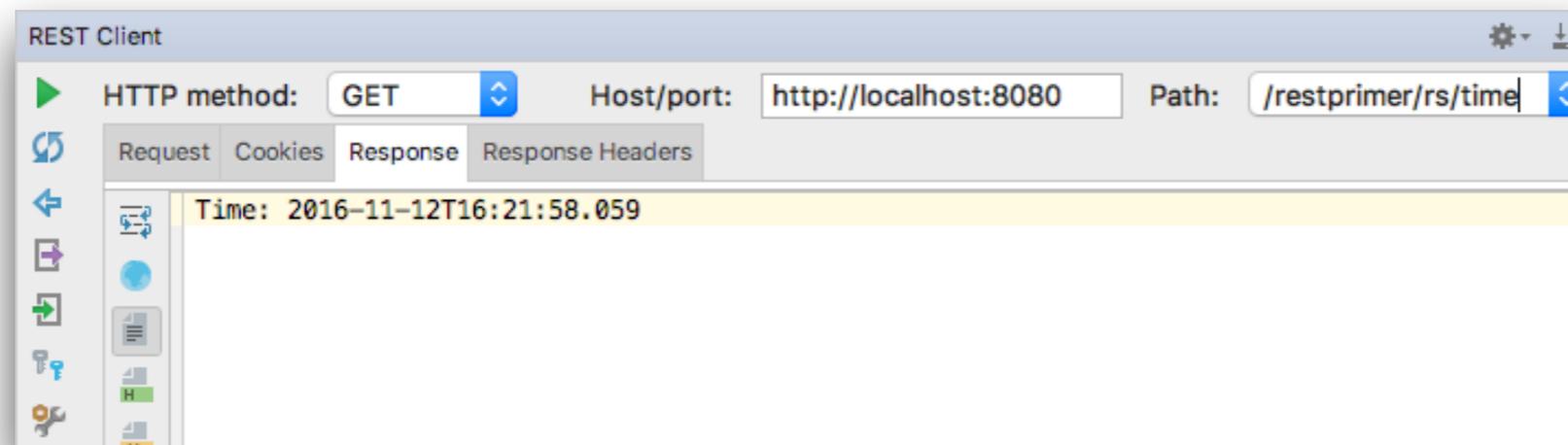
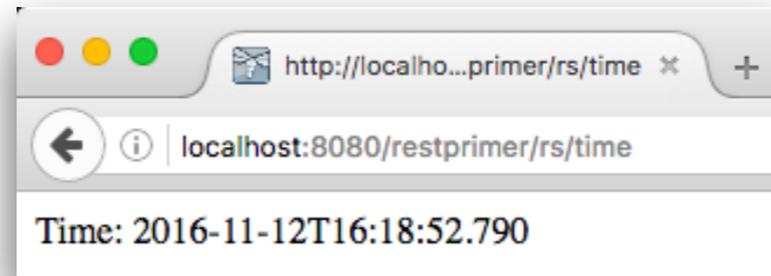
Verbesserung der Ausgabe

Wie muss hier die Rückgabe aussehen?

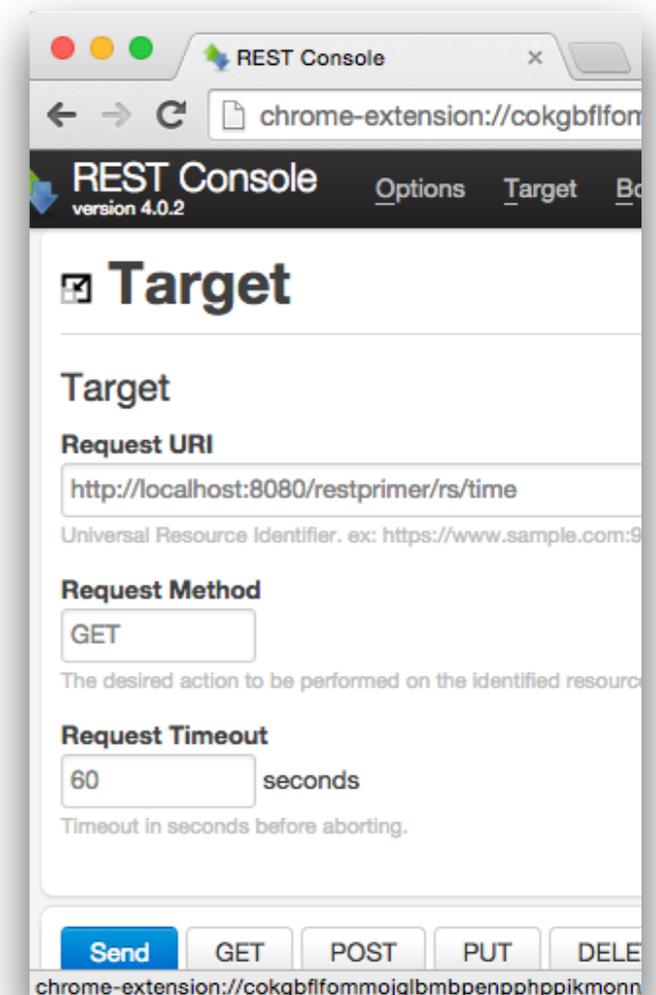


Mögliche Clients

- Web-Browser
- REST Client in IDE



- Plugins in WebBrowser
oder Standalone REST-Clients





File Authentication Headers View Favorite Requests Setting **RESTClient**

[-] Request

Method URL

[-] Response

Response Headers **Response Body (Raw)** Response Body (Highlight) Response Body (Preview)

28. August 2015, 09:10:38

Hier sieht man sehr gut die Header-Informationen wie Status Code und Content-Type

[-] Response

Response Headers **Response Body (Raw)** Response Body (Highlight) Res

1.	Status Code	: 200 OK
2.	Connection	: keep-alive
3.	Content-Length	: 25
4.	Content-Type	: text/html
5.	Date	: Fri, 28 Aug 2015 07:10:38 GMT
6.	Server	: WildFly/9
7.	X-Powered-By	: Undertow/1

Eigener Java-Client

- Der Nachteil der vorher genannten Clients besteht darin, dass die REST-Abfragen i.N. nicht wiederholbar sind. D.h. bei jeder Änderung im Code, sind die Tests manuell durchzuführen
- Abhilfe: Die Erstellung eines eigenen Java-Clients (mit der Jersey API, REST assured, ...)

Exkurs: Java Community Process (~~JCP~~) EFSP

- Die Sprache Java wird durch den *Java Community Process* bzw den *Eclipse Foundation Specification Process* weiterentwickelt
- Mitglieder des JCP (Firmen und Einzelpersonen) können Vorschläge zur Änderung der Sprache Java einbringen → Java Specification Request (JSR)
- Wird ein JSR angenommen, erstellt eine Expertengruppe eine Referenzimplementierung
- Für JAX-RS (Java API for RESTful Web Services) ist dies **Jersey**.
- Weitere Implementierungen sind zB RESTEasy, CXF, Restlet,...

Eclipse Foundation Specification Process

- EFSP
- https://www.eclipse.org/community/eclipse_newsletter/2019/january/EFSP_vs_JCP.php
- <https://jaxenter.de/ee-insights-jakarta-ee-8-spezifikationsprozess-82216>
- <https://jaxenter.de/5-unterschiede-jcp-efsp-78514>

REST assured - Testclient

The image shows an IDE window with the following elements:

- Project Structure:** A tree view on the left showing the project hierarchy: `getting-started` (root) contains `.idea`, `.mvn`, `http-requests`, `src` (subfolders: `main`, `test`), and `resources` (subfolders: `META-INF`, `application.properties`). The `main` folder contains `docker`, `java` (subfolders: `at`, `getting`, `started`), and `target`. The `started` folder contains `GreetingResource` and `TimeServerResource`.
- Code Editor:** The main editor shows the `TimeServerResource.java` file with the following code:

```
1 package at.htl.getting.started;
2
3 import javax.ws.rs.GET;
4 import javax.ws.rs.Path;
5
6 import java.time.LocalDateTime;
7
8 @Path("time")
9 public class TimeServerResource {
10     @GET
11     public String time() {
12         return "Time: " + LocalDateTime.now();
13     }
14 }
15
```
- Context Actions:** A green callout box over the class name says: **Show Context Actions** via `\↵` (`Alt+↵` for Win/Linux).
- Context Action Menu:** A menu is open over the class name with options: `Create Test`, `Make 'TimeServerResource' package-private`, `Create...`, and `Add...`.
- Create Test Dialog:** A dialog box titled "Create Test" is open. It has the following fields:
 - Testing library: `JUnit5`
 - Class name: `TimeServerResourceTest`
 - Superclass: (empty)
 - Destination package: `at.htl.getting.started`
 - Generate: `setUp/@Before`, `tearDown/@After`
 - Generate test methods for: `Show inherited methods`
- Test Method List:** Below the dialog, a table lists the test methods to be generated:

Member
<input type="checkbox"/> <code>m</code> <code>time():String</code>

REST assured ist in pom.xml bereits konfiguriert

The screenshot displays an IDE window for a project named 'getting-started'. The main editor shows the source code for `TimeServerResourceTest.java`. The code is as follows:

```
1 package at.htl.getting.started;
2
3 import org.junit.jupiter.api.Test;
4
5 import static io.restassured.RestAssured.when;
6 import static org.hamcrest.Matchers.startsWith;
7
8 class TimeServerResourceTest {
9
10     @Test
11     public void fetchTime() {
12         when() RequestSender
13             .get(s: "time") Response
14             .then() ValidatableResponse
15                 .statusCode(200)
16                 .body(startsWith("Time:"));
17     }
18 }
```

Below the code editor, the 'Run' tab shows the test execution results. The output indicates that the test passed successfully:

```
Run: TimeServerResourceTest x
Tests passed: 1 of 1 test - 1 s 118 ms
Test Results 1 s 118 ms
  TimeServerResou 1 s 118 ms
    fetchTime() 1 s 118 ms
Process finished with exit code 0
```

Two callout boxes are present: a green one in the top right corner with the text 'Test funktioniert' (Test works), and a blue one in the bottom right corner with the text 'Vewendung von Hamcrest' (Usage of Hamcrest).

Test soll auch fehlschlagen

The screenshot shows an IDE window titled "getting-started - TimeServerResourceTest.java". The main editor displays the following Java code:

```
1 package at.htl.getting.started;
2
3 import org.junit.jupiter.api.Test;
4
5 import static io.restassured.RestAssured.when;
6 import static org.hamcrest.Matchers.startsWith;
7
8 class TimeServerResourceTest {
9
10     @Test
11     public void fetchTime() {
12         when() RequestSender
13             .get(s: "time") Response
14             .then() ValidatableResponse
15                 .statusCode(200)
16                 .body(startsWith("xTime:"));
17     }
18 }
```

The IDE interface includes a Project Structure view on the left showing the package structure, a Run window at the bottom, and a Test Results window. The Run window shows "Tests failed: 1 of 1 test - 1 s 129 ms". The Test Results window shows a failed test "fetchTime()" with the following error message:

```
java.lang.AssertionError: 1 expectation failed.
Response body doesn't match expectation.
Expected: a string starting with "xTime:"
Actual: Time: 2020-08-29T13:53:41.878498
```

The IDE status bar at the bottom indicates "Tests failed: 1, passed: 0 (moments ago)", "16:36", "LF", "UTF-8", "4 spaces", and "master".

Lösung

```
package at.htl.restprimer.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

@Path("time")
public class TimeServerResource {

    @GET
    public String time() {
        return "Time: " + LocalDateTime
            .now()
            .format(DateTimeFormatter.ofPattern("dd. MMMM yyyy, hh:mm:ss"));
    }
}
```