LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Praktikum Mobile und Verteilte Systeme

RESTful web services & mobile push architectures

Prof. Dr. Claudia Linnhoff-Popien
Philipp Marcus, Mirco Schönfeld

Sommersemester 2015

RESTful web services & mobile push – Introduction

Today:

- RESTful web services
- Comet programming techniques
- Mobile push services
  - C2DM/GCM
  - APNS
  - …
- and how to use all these in Java and Android

Next week:

- Context-sensitive services

Representational State Transfer (REST)

- A lightweight alternative to the SOAP/WSDL universe

- Defined by Roy T. Fielding
  - Dissertation: “Architectural Styles and the Design of Network-based Software Architectures“ (2000)
  - main author of HTTP/1.0 and HTTP/1.1
  - co-founder of the Apache HTTP server project (**httpd**)
- REST is an architectural style (and HTTP can be regarded as one incarnation of it)
- REST relies on some important architectural principles:
  - Everything is a **resource**
  - Communicate **statelessly**
  - Use a **common interface** for all resources
  - Resources can have **multiple representations**

REST principles I: Everything is a resource

- from a REST point of view, **every data element** of an application a designer deems worthy of having its own URI **is a resource**
  - entities, attributes, collections, etc.
- each resource has a **unique ID**
  - REST makes use of a resource's URI
    - global standard namespace, globally unique
- a resource is not an actual object or service itself, but rather **an abstract interface** for using it
- using **human-readable URIs** is common (yet not obligatory)

```
http://example.com/customers/1234
http://example.com/orders/2013/1/12345
http://example.com/orders/2013/1
http://example.com/products/4554
http://example.com/products?color=green
http://example.com/processes/salary-increase
```

REST principles II: Communicate statelessly

- REST includes the concept of **statelessness** on behalf of the server
  - but, of course, there is some state…
- All application state should either
  - be **turned into resource state**
  - or be **managed at the client**
- All **requests should be independent** from earlier requests
  - messages are self-contained, including all necessary information
- Advantages:
  - **scalability**
  - **isolation of the client** against changes on the server

REST principles III: Use standard methods

- REST demands the usage of **simple, uniform interfaces** for all resources

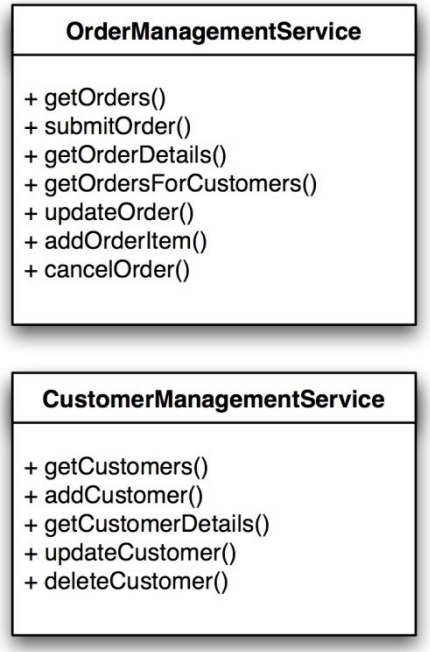
- When making a HTTP request on a resource, we expect the application to actually **do something meaningful**
  - this is achieved with every resource providing the same interface (i.e., the same set of methods)

- REST is making usage of the **HTTP verbs** (as in the HTTP specification)
- With REST, these verbs are mapped to resource-specific semantics

```
class Resource {                        // analogy to oo-programming
  Resource(URI u);                      // URI
  Response get();                       // HTTP GET
  Response post(Request r);             // HTTP POST
  Response put(Request r);              // HTTP PUT
  Response delete();                    // HTTP DELETE
}
```

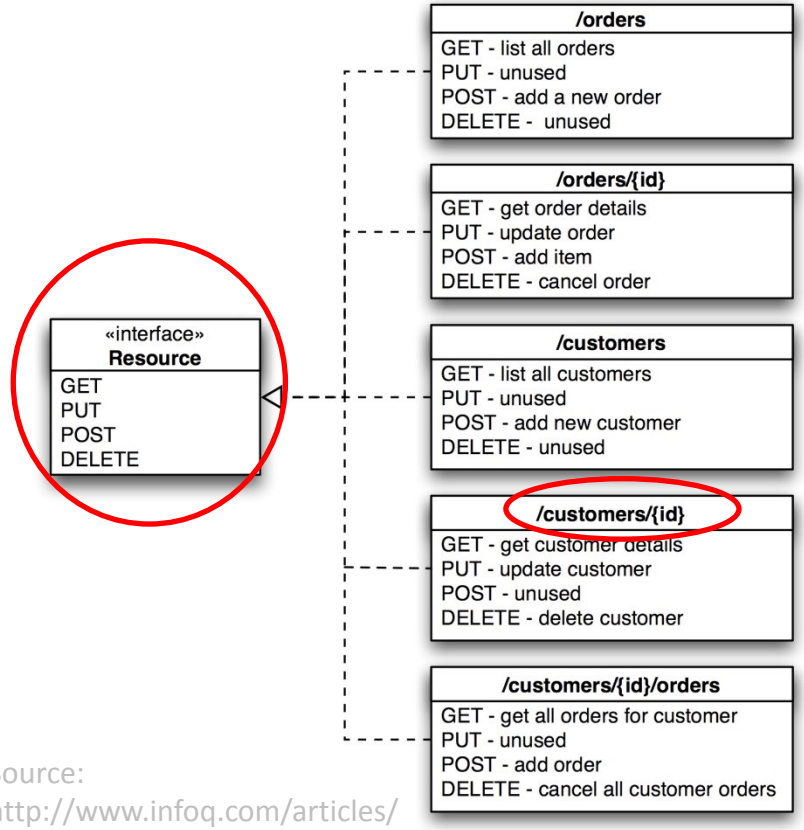
An Example of a RESTful Webservice

Client needs to know and handle the services‘ interfaces:



Source:
http://www.infoq.com/articles/rest-introduction

RESTful HTTP design:
Any HTTP client can be used to access the services



Source:
http://www.infoq.com/articles/rest-introduction

REST principles IV: Different representations

- Resources can (and actually should) have **multiple representations**
  - provide multiple representations of resources for different needs
  - ideally, at least one standard format should be provided

- Selection of data formats is done **using HTTP content negotiation**
  - clients can ask for a representation in a particular format

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/xml
```

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

- Advantages:
  - Having several representations of a resource (e.g., text, XML, HTML, JSON…), they are consumable by standard web browsers
  - An application's Web UI can actually be regarded as its Web API, providing a better Web interface for both humans and applications

REST-conformant usage of HTTP methods

- **HTTP GET**
  - Used for accessing the requested resource without any side-effects. A resource must never be changed via a GET request (read-only)!
- **HTTP PUT**
  - Used for creating or updating a resource at a known URI.
- **HTTP DELETE**
  - Used for removing a resource.
- **GET**, **PUT** and **DELETE** must be implemented as idempotent methods
  - can be called repeatedly without leading to different results
- **HTTP POST**
  - Update an existing resource or create a new one (not idempotent)

A simple example of a RESTful web service

- Mapping of “normal” method names to RESTful resource interfaces
  - combination of resource URIs and the standard HTTP methods

Normal method name	URI (RESTful resource)	HTTP method
listOrders	/orders	GET
addNewOrder	/orders	POST
addNewOrder	/orders/12344	PUT
getOrder	/orders/12344	GET
deleteOrder	/orders/12344	DELETE
listCustomers	/customers	GET
getCustomer	/customers/dorfmeister	GET
addCustomer	/customers	POST
addCustomer	/customers/marcus	PUT
updateCustomer	/customers/dorfmeister	PUT
…		

Advantages of the RESTful approach

- **Simplicity**
  - well known interfaces (URIs, HTTP methods), no new XML specification
- **Lightweightness**
  - short messages, little overhead
- **Multiple representations**
- **Security**
  - authentication and authorization can be done by the web server
- **Scalability/Reliability**
- **Caching**
- **Easy service orchestration** (via hyperlinks)
  - URIs define global namespace, no application boundaries

REST vs. SOAP (1)

- requesting a user's details **using SOAP** (via a POST request)

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:body pb="http://www.example.com/phonebook">
    <pb:GetUserDetails>
      <pb:UserID>12345</pb:UserID>
    </pb:GetUserDetails>
  </soap:Body>
</soap:Envelope>
```

- requesting a user's details **using REST** (via a GET request)

```
http://www.example.com/phonebook/UserDetails/12345
```

- REST resources are usually **defined as nouns**, not as verbs
  - GetUserDetails (SOAP) vs. UserDetails (REST)

REST vs. SOAP (2)

- In contrast to Service oriented architectures (such as SOAP), REST can be considered a Resource Oriented Architecture (ROA)

	RESTful Web services	SOAP Web services
Architectural style	REST/ROA	SOA
Server state	Stateless	Stateless or stateful
Data format	Text, HTML, XML, JSON, binary, …	XML
Application Protocol	REST	SOAP
Level of formality of interface definitions	Rather low (XSD, WADL) (not specified)	High (WSDL)
Typing	None	Strong
Support for asynchronous communication	No	Yes (WS-Notification)
Caching of results	yes	no
Scalability	high	medium
Performance	high	lower
ACID transactions	no	Yes (WS-AtomicTransaction)
Access control	Webserver (easy)	WS-Security (more complex, yet more powerful)
Fields of application	Data-oriented, short term services	Both data-oriented and long-term process-oriented services

REST vs. SOAP (3)

- **REST**
  - is **easy to understand**
  - offers maximum **performance and scalability**
  - makes use of **existing standards** only (i.e., URI and HTTP)
  - is perfectly fit for handling **CRUD operations** on data using a single common interface

- **SOAP**
  - brings its **own protocol**
  - focuses on **exposing application logic** (not resources) as a service using different interfaces
  - is supported by a plethora of existing **software tools**
  - allows for **ACID transactions** (WS-AtomicTransactions), mature **security mechanisms** (WS-Security) and **guaranteed message delivery** (WS-ReliableMessaging) → enterprise security features

Using REST with Java: Jersey (JAX-RS)

- Jersey is “the open source, production quality, JAX-RS (JSR 311) Reference Implementation for building RESTful Web services”
- can be downloaded from **jersey.java.net** 
- works with any Servlet Container (e.g., Apache Tomcat or Grizzly)
- contains both server and client APIs
- Jersey supports the automatic creation (marshalling) of XML and JSON representations of resources (based on JAXB)
- as a key feature, JRS 311 makes use of **Java annotations** to define the REST relevance of Java classes (Media-Type, HTTP-Method, URI, …)

Getting started with Jersey and Apache Tomcat

- Create a new Dynamic Web Project, then download and copy the Jersey JAR-files to `WEB-INF/lib`

- After installing and configuring Tomcat, applications can be deployed…
  - by copying the application's `WEB-INF/` and `META-INF/` folders to a subfolder of Tomcat's `webapps` directory
  - by creating a WAR (**W**eb **AR**chive)-file of the application and storing it in the `webapps` folder

- Every web application should include a deployment descriptor (according to the Servlet 2.4 specification)
  - this file (`web.xml`) must always be placed in the `WEB-INF/` folder
  - can be generated using Eclipse (but has to be modified)

- Deployed Webapps can be managed (start, stop, reload, etc.) using Tomcat's Application Manager, accessible at **http://host:port/manager**

Example web.xml file

- In order to correctly dispatch incoming requests to the Jersey servlet, the RESTful application's `web.xml` should look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi= … >
 <display-name>DISPLAY_NAME</display-name>
 <servlet>
  <servlet-name>SERVLET_NAME</servlet-name>
  <servlet-class>
     com.sun.jersey.spi.container.servlet.ServletContainer
  </servlet-class>
  <init-param>
   <param-name>com.sun.jersey.config.property.packages</param-name>
   <param-value>your.package.name</param-value>
  </init-param>
 </servlet>
 <servlet-mapping>
  <servlet-name>SERVLET_NAME</servlet-name>
  <url-pattern>/whatever/you/want/*</url-pattern>
 </servlet-mapping>
</web-app>
```

Using POJOs for building RESTful Web services

- with Jersey, RESTful web services can be realized by simply annotating POJOs in order to define allowed HTTP-methods, content-types, parameters, etc.
- such classes are known as **root resource classes**

```
@Path("/hello") //the resource's URI
public class Hello {

 @GET //HTTP method
 @Produces(MediaType.TEXT_PLAIN) //requested content-type
 public String sayHello() {
   return "Hello Jersey";
 }

 @GET //HTTP method
 @Produces(MediaType.TEXT_XML) //requested content-type
 public String sayXMLHello() {
   return "<?xml version=\"1.0\"?>" + "<hi>Hello Jersey" + "</hi>";
 }
}
```

Annotation basics in JAX-RS (1)

- the **@Path** annotation's value is a relative URI path. The base URI is the application path as set in the `web.xml` (display-name + url-pattern).

- *Resource method designator* annotations
  - **@GET** can be used to read a resource without any side effects(!)
  - **@POST** creates a new resource (not idempotent)
  - **@PUT** creates or modifies an existing resource (idempotent)
  - **@DELETE** removes an existing resource (idempotent)
  - **@HEAD** returns the same as @GET, just without the body

- **@Consumes** specifies the MIME types of representations a resource can consume from a client

- **@Produces** specifies the MIME types of representations a resource can produce and send back to a client

Annotations in JAX-RS (2)

- **@Path** annotations might (or might not) have a leading or ending '/' , it doesn't make a difference:
  - a leading '/' in a path is ignored
  - base URIs are treated as if they ended in '/'

- A resource with relative **@Path** annotation can be found at `http://host:port/<display-name>/<url-pattern>/<@Path>`

- **@Consumes** and **@Produces** can be applied at class and method levels

- More than one media type may be declared in the same **@Produces** or **@Consumes** declaration

- Method level annotations can be used to **override** class level annotations

Useful features of JAX-RS (1)

- One thing that makes Jersey extremely useful is that you can embed variables in the URIs (so-called *URI path templates)*:

```
@Path("/login/{user}") //{user} will be substituted
public class UserResource {

 @GET //HTTP method
 @Produces("text/plain") //output format
 public String loginUser(@PathParam("user") String userName){
   return "hi, "+userName;
 }
}
```

- and naturally also as query parameters:

```
@Path(„/foo") //simply another path…
@GET //HTTP method
public Response bar(@DefaultValue("1") @QueryParam("a") int a,
      @DefaultValue("true") @QueryParam("b") boolean b) {
  …
}
```

Useful features of JAX-RS (2)

- If the HTTP request contains a body (PUT, POST requests), this data can easily be accessed as a method parameter:

```
@POST //HTTP method
@Consumes("text/plain") //input format of request body
 public String handlePlaintext(String message){
    //store the plaintext message somewhere
    …
 }
}
```

- If several methods exist for the same resource, Jersey will select the (most) appropriate one for handling a request (method, MIME types…)

```
@POST //HTTP method
@Consumes(MediaType.TEXT_XML) //input format of request body
 public String handleXML(String message){
    //store the xml string somewhere
    …
 }
}
```

Jersey and JAXB

- Jersey allows for the automatic mapping (marshalling) from POJOs to representations in XML (and also JSON!)
- Realized with the support of JAXB (*Java Architecture for XML Binding):*
  - Java standard defining how to convert Java objects from/to XML
  - provides a standard set of mappings
  - defines an API for reading and writing Java objects to and from XML
  - JAXB is making usage of Java annotations, too

```
//Define the root element for a XML tree
@XmlRootElement(namespace = "namespace")
//Set the order of the fields in the XML representation
@XmlType(propOrder = { "field2", "field1",.. })
//generate a XML wrapper element
@XmlElementWrapper(name = "wrapper_element")
//set the name of the entities
@XmlElement(name = "element_one")
```

Example of Jersey using JAXB (1)

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("")
public class TicketServer {

        /**
         * Returns the server status.
         *
         * @return the server status
         */
        @GET
        @Path("ping")
        @Produces(MediaType.APPLICATION_JSON)
        public ServerStatus getPing() {
                return ServerStatus.getServerStatusInstance();
        }
}
```

Example of Jersey using JAXB (2)

```
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.Date;

@XmlRootElement
public class ServerStatus {
  private final static ServerStatus instance = new ServerStatus();

  public static ServerStatus getServerStatusInstance() {
    return instance;
  }

  private boolean running; //Laufzeitstatus

  @XmlElement(name = "running")
  public boolean isRunning() { return running; }

  @XmlElement(name = "server_now")
  public Date getServerNow() { //Aktuelle Zeit
    return new Date();
  }
  …
}
```

Example of Jersey using JAXB (3)

- Testing the web service in your browser
  - Request:

```
GET /ping HTTP/1.1
Host: example.com
Accept: application/json
```

  - Response:

```
{ "running": "true",
  "server_now": "2013-01-10T16:31:56.843+01:00“,
  … }
```

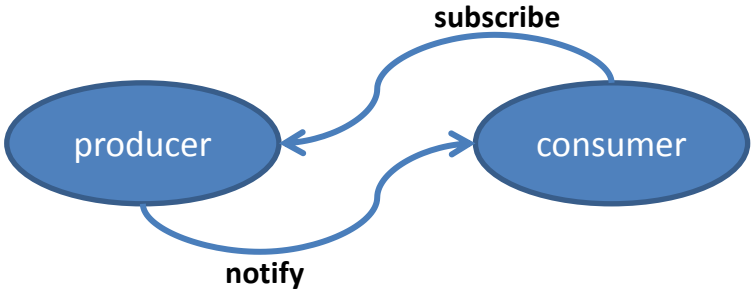
Asynchronous communications

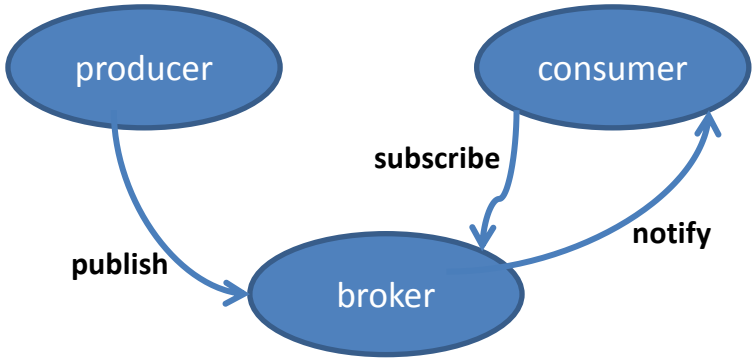
How to notify clients about changed resources or updates?

More general: How to **handle server-side events asynchronously**?

- **polling** is ineffective (e.g., continuously requesting a web service)

- SOAP offers **WS-Notification**
  - either peer-to-peer or brokered





- **Comet programming**: strategies for realizing push-like communication in pull-based environments (using HTTP)

Comet programming

- A web application model using persistent HTTP requests to push data to a browser
- Term coined by software engineer Alex Russell in a blog post in 2006
- First implementations date back to 2000
  - Pushlets, Lightstreamer, KnowNow
- In 2006, some widely known applications adapted these techniques
  - web-based chat application for AOL, Yahoo, Microsoft chat (Meebo)
  - Google: integration of a **web-based chat** in GMail
  - Comet-based, real-time **collaborative document editing** (JotSpot)

- Comet is an umbrella term, encompassing multiple techniques
  - relying on features included by default in browsers (e.g., JavaScript)
  - also known as Ajax Push, Reverse Ajax, Two-way-web, HTTP Streaming

Comet implementations

- **Streaming-based** implementations
  - Hidden iframe
    - uses chunked transfer encoding (no content-length) containing JavaScript tags
    - working in every common browser
  - XMLHttpRequest
    - server sends “multipart HTTP response” with each part invoking `onreadystatechange` callback
    - only working with few browsers

- **Long-polling** based implementations
  - XMLHttpRequest long polling
    - works like the standard use of XHR
    - an asynchronous request is sent to the server, response only after an update
    - after processing the response (or after a timeout), a new request will be sent
  - Script tag long polling
    - dynamically create script elements as `<src="cometserver/…js">`
    - payload contains new JavaScript events
    - cross-browser and cross-domain functionality

Mobile push architectures

- **Push notifications…**
  - are messages pushed to a central location and delivered to mobile devices
  - are comparable to the publish/subscribe pattern
  - often contain other technologies such as alerts, tiles, or raw data
  - offer an alternative to constantly polling data from servers

- These “central locations” are nowadays provided by Google, Apple, Microsoft, Blackberry, …

- Goal: **Push, don’t pull**
  - only fetch data when useful

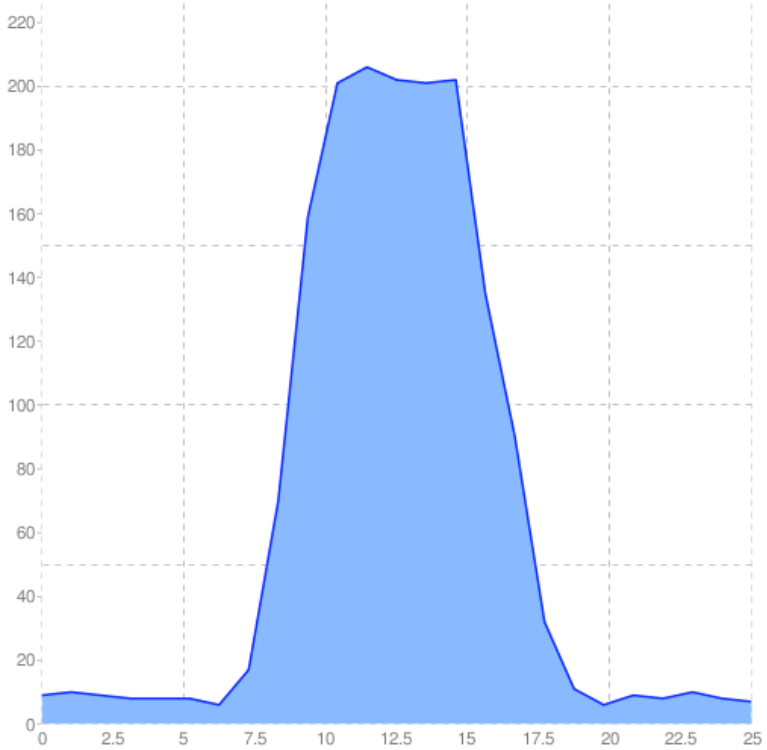
Advantages of push notifications (1)

Battery Life

- Baseline: 5-8 mA
- Network: 180-200 mA
  - Tx is more expensive than Rx
- Radio stays on for few seconds
- 0.50 mAh for a short poll
  - 5m frequency: ~144 mAh / day
  - 15m frequency: ~48 mAh / day

- Push notification services are running in the background

- Pushing data is hence **more effective** than polling, if #updates < #polls



Source: Android development team at Google

Advantages of push notifications (2)

- **Message delivery and „time of flight“**
  - to save on battery, polls are usually spaced 15+ minutes apart
  - updated data might hence also be 15+ minutes late!
  - when using push notifications, message delivery can usually be expected to be a matter of seconds (<5s)
  - push notifications can also be sent to a currently offline device

- However, generally there is **no guarantee for delivery**
  - one might exceed quotas
  - some notification servers only allow a single message to be in queue at a time
  - …

Google C2DM

- The Cloud to Device Messaging framework allowed third-party servers to send lightweight messages to corresponding Android apps
- Designed for notifying apps about new content
- Makes **no guarantees** about delivery or the order of messages.
- Apps **do not have to be running** to receive notifications
  - the system will wake up the application via an Intent broadcast
- only passes raw data received to the application
- Requirements:
  - devices running Android 2.2 or above
  - have the Market application installed
  - a logged in Google account

- launched in 2010, officially deprecated as of June 26, 2012!
  - existing apps are still working, though

Google Cloud Messaging (GCM)

- successor of G2DM
- main differences:
  - to use the GCM service, you need to **obtain a Simple API Key** from the Google APIs console page
  - in C2DM, the Sender ID is an email address. In GCM, the **Sender ID** is a project number (acquired from the API console)
  - GCM HTTP requests **support JSON format** in addition to plain text
  - In GCM you can send the same message to multiple devices simultaneously (**multicast messaging**)
  - **Multiple parties** can send messages to the same app with one common registration ID
  - apps can send expiring invitation events with a **time-to-live** value between 0 and 4 weeks
    - GCM will store the messages until they expire
  - "**messages with payload**" to deliver messages of up to 4 Kb
  - GCM will store up to 100 messages
  - GCM provides **client and server helper libraries**

Google Cloud Messaging architecture (1)

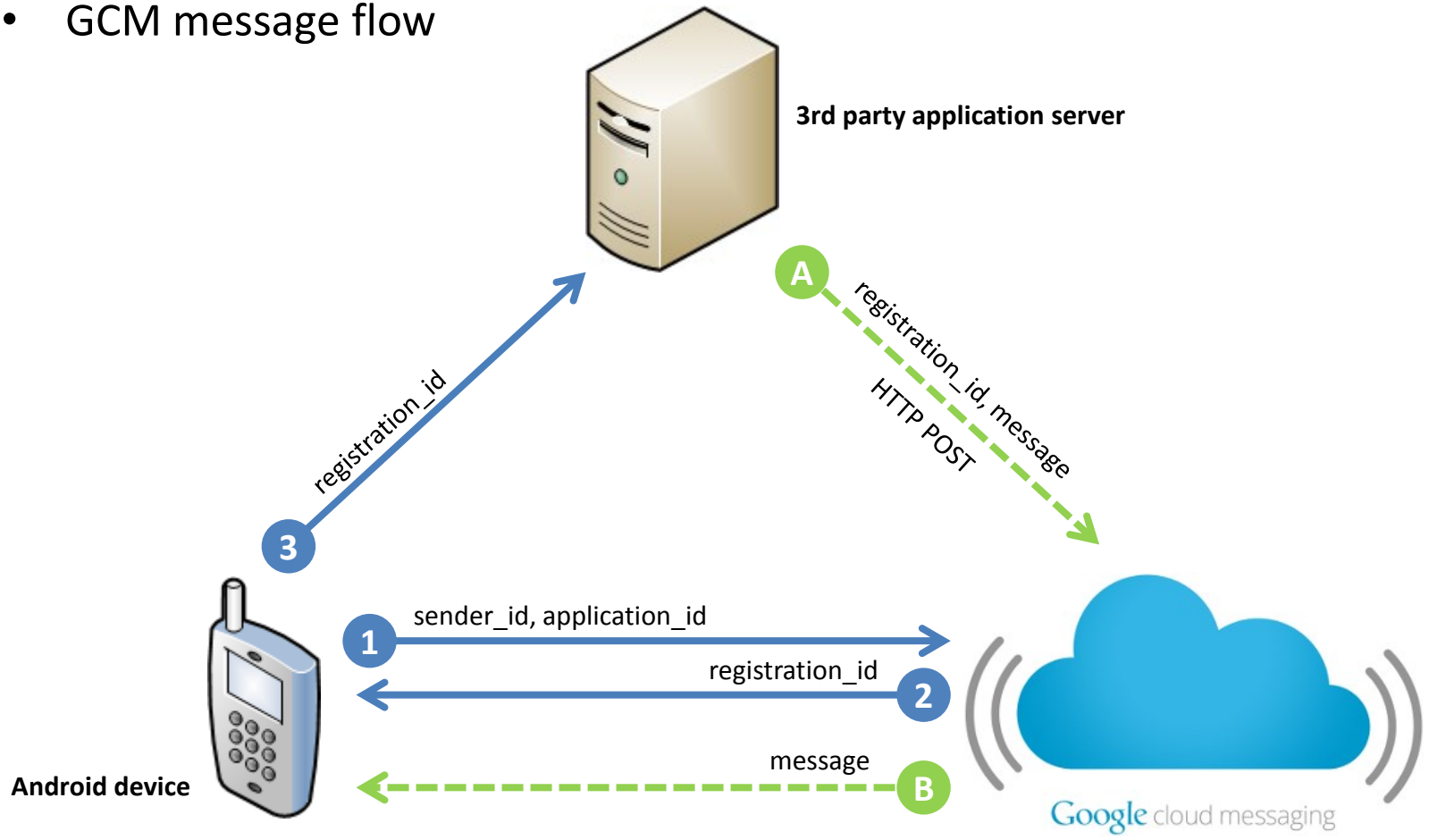
- GCM components
  - **Mobile Device** 
    - running an Android application that uses GCM
    - must be a 2.2 Android device that has Google Play Store installed
    - must have at least one logged in Google account
  - **3rd-party Application Server** 
    - a server set up by an app developer as part of implementing GCM
    - sends data to an Android application on the device via GCM
  - **GCM Servers** 
    - the Google servers involved in taking messages from the 3rd-party application server and sending them to the device

Google Cloud Messaging architecture (2)

- Credentials used in GCM
  - **Sender ID**
    - the project number (acquired from the API console)
    - used for 3rd party server in order to ensure that the account is permitted to send messages to the GCM servers
  - **Application ID**
    - used for identifying the application that is registering to receive messages (its package name as in the manifest file)
  - **Registration ID**
    - issued by the GCM servers to the Android application
    - used for identifying devices on the 3rd party server
  - **Google User Account**
  - **Sender Auth Token** (API key)
    - an API key stored on the 3rd-party application server
    - grants the application server authorized access to Google services

Google Cloud Messaging architecture (3)

- GCM message flow



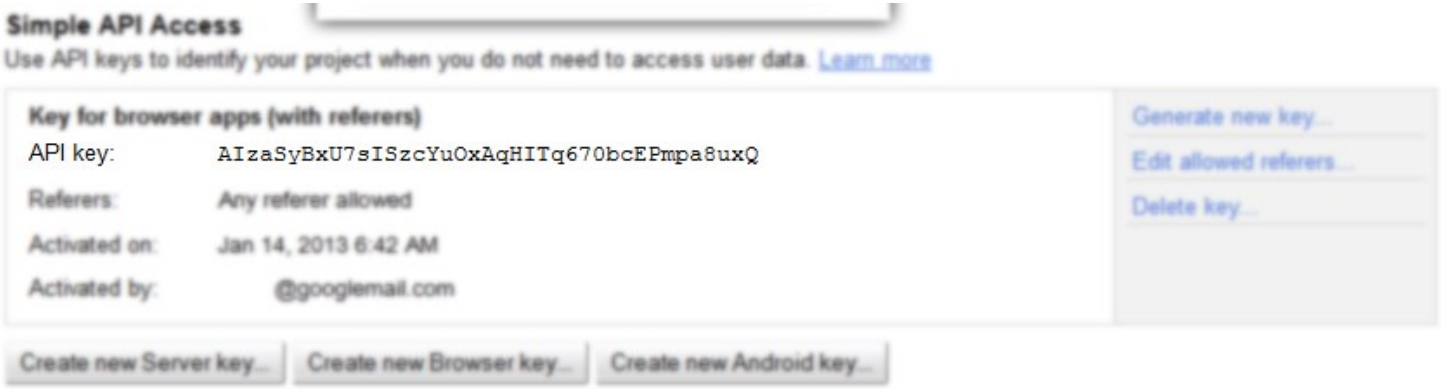
Using GCM with Java and Android (1)

- Create a **new Google API project** in order to get your SENDER_ID
  - Google APIs Console https://code.google.com/apis/console
  - https://code.google.com/apis/console/#project:**XXXXXXXXXX**

SENDER_ID

- **Enable GCM** services
  - Services → Google Cloud Messaging → ON

- Generate and find your **API key** (IP table might be empty)



Using GCM with Java and Android (2)

- Writing the **client application**
  - Download the **helper libraries**
    (SDK Manager, Extras > Google Cloud Messaging for Android Library)
  - Copy **gcm.jar** to your application's classpath
  - Adapt the Android **manifest file:**
    - **minSdkVersion** must be 8 or above
    - declare and use a **custom permission**, so that only your app will receive your push messages

```
<permission android:name="my_package.permission.C2D_MESSAGE"
        android:protectionLevel="signature" />
<uses-permission
        android:name="my_package.permission.C2D_MESSAGE" />
```

    - add **further permissions**:
      - `com.google.android.c2dm.permission.RECEIVE`
      - `android.permission.GET_ACCOUNTS`
      - `android.permission.WAKE_LOCK`

Using GCM with Java and Android (3)

- Writing the **client application**
  - add a broadcast receiver entry for `com.google.android.gcm.GCMBroadcastReceiver` (provided by the GCM library)

```
<receiver android:name="com.google.android.gcm.GCMBroadcastReceiver"
android:permission="com.google.android.c2dm.permission.SEND" >
  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.RECEIVE" />
    <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
    <category android:name="my_package" />
  </intent-filter>
</receiver>
```

  - add a `<service/>` entry for `.GCMIntentService`
  - implement `GCMIntentService` as subclass of `GCMBaseIntentService`
    - override at least its `onRegistered(), onUnregistered(), onMessage()` methods in order to be able to react to notifications

Using GCM with Java and Android (4)

- Writing the **client application**
  - handle notifications in the `onReceive` method

```
@Override
protected void onMessage(Context context, Intent intent) {
  String message = intent.getStringExtra("message");
  … // create a local notification (e.g., in the status bar)
}
```

  - in your main Activity, add something similar to this:

```
GCMRegistrar.checkDevice(this);
GCMRegistrar.checkManifest(this);
final String regId = GCMRegistrar.getRegistrationId(this);
if (regId.equals("")) {
  GCMRegistrar.register(this, SENDER_ID);
} else {
  Log.v(TAG, "Already registered");
}
```

Using GCM with Java and Android (5)

- Writing the **server-side application**
  - copy the **gcm-server.jar** to your server classpath
  - provide interfaces for registering and unregistering of devices
    - upon registration, a devices **registrationId** has to be stored
  - implement functionality for sending notifications to the registered devices when needed

```
Sender sender = new Sender("AIzaXXXXXXXXXXXXXXXXXXXXXXXXX");

Message message = new Message.Builder()
.collapseKey("1")
.timeToLive(3)
.delayWhileIdle(true)
.addData("message","sample text!")
.build();

Result result = sender.send(message,"device_token", 1);
```

API_KEY

REGISTRATION_ID

#retries

RESTful web services & mobile push – Practical

- What you will do:
  - Build a simple event calendar as a RESTful webservice using Jersey
  - Implement a client application for Android for displaying, adding (and maybe also editing) calendar entries

- Bonus:
  - Utilize GCM for pushing update notifications to the client app