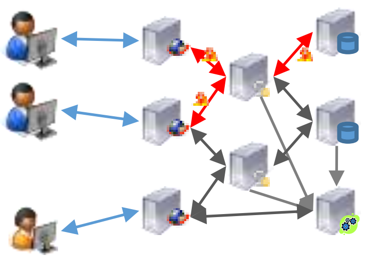


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

6

Recommended Lectures

Apache Spark:
Machine Learning (ML) with Spark

Advanced Data Analytics …

- .. Are techniques for *deriving insights* and *making predictions* or *recommendations* based on data

Common tasks include:

- Supervised learning: classification and regression
  - Goal: predict label/real value for data point based features
- Recommendation engines
  - Goal: suggest products to users based on behavior
- Unsupervised learning: clustering, anomaly detection, topic modeling
  - Goal: discover structure in the data
- Graph analytics: finding patterns in networks

MLlib: a Core Package of Spark

- **MLlib** is a package of Spark with APIs/routines for:
  - Gathering and cleaning data
  - Feature engineering and feature selection
  - Training and tuning large-scale un/supervised ML models
  - Using such ML models in production
- **MLlib** consists of two packages
  - **pyspark.ml** (or org.apache.spark.ml)
    - Preferred higher level API, currently the “main” API (Spark 2.x)
    - Uses DataFrames and provides ML pipelines
  - **pyspark.mllib** (or org.apache.spark.mllib)
    - Lower-level package using RDDs
    - Now in maintenance mode, no new features

Workflow for Obtaining ML Models in Spark

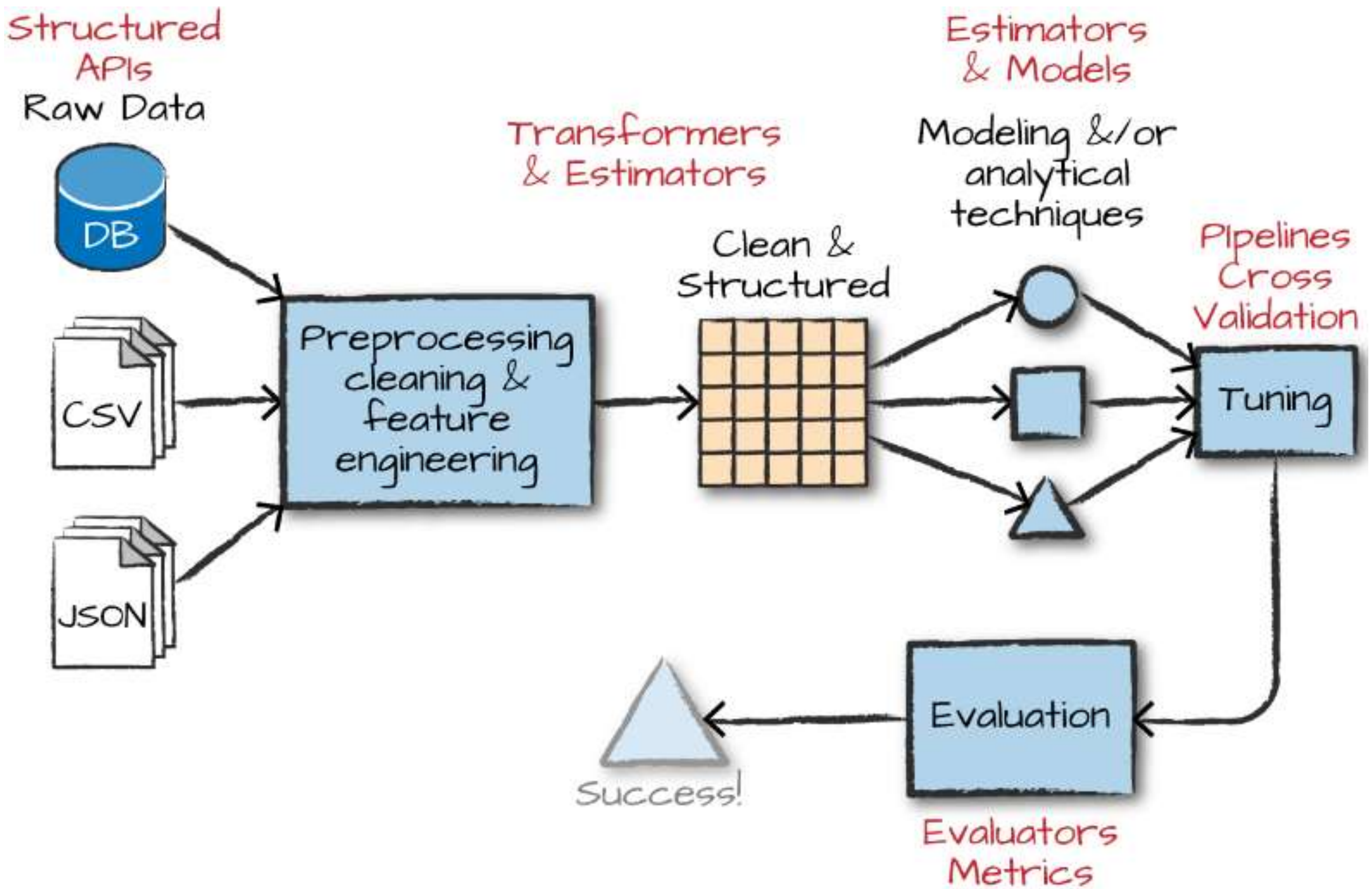


Fig. 24-2. from *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

ML Pipelines in Spark

- Spark's **ML Pipelines** API (link) allow to set up a sequence of stages for
  - Data cleaning, feature extraction, model training, model validation and tuning, model testing, ..
- They make it easier to combine multiple algorithms into a single pipeline (workflow)
  - The pipeline concept is mostly inspired by the scikit-learn project (for Python)
- They use DataFrames (DF), i.e. essentially DB-like tables with columns of various types
  - See previous lecture

Essential Elements of the ML Pipelines API

- **Transformer**: an algorithm which can transform one DataFrame into another DataFrame
  - E.g., an ML model is a T. which transforms a DataFrame with features into a DataFrame with predictions
- **Estimator**: an algorithm which can be fit on a DataFrame to produce a Transformer
  - E.g., a learning algorithm is an E. which trains on a DataFrame and produces a *model*
- **Pipeline**: an object which chains multiple Transformers and Estimators together to a workflow
- **Parameter**: a common API for specifying parameters of Transformers and Estimators

Transformers: Details

- An abstraction that includes *feature transformers* and *learned models*
- Transformers convert data in some way
  - E.g. normalize a column; predict a label for each feature
- They add more columns or change the DF values
- Call the method **transform()** to activate

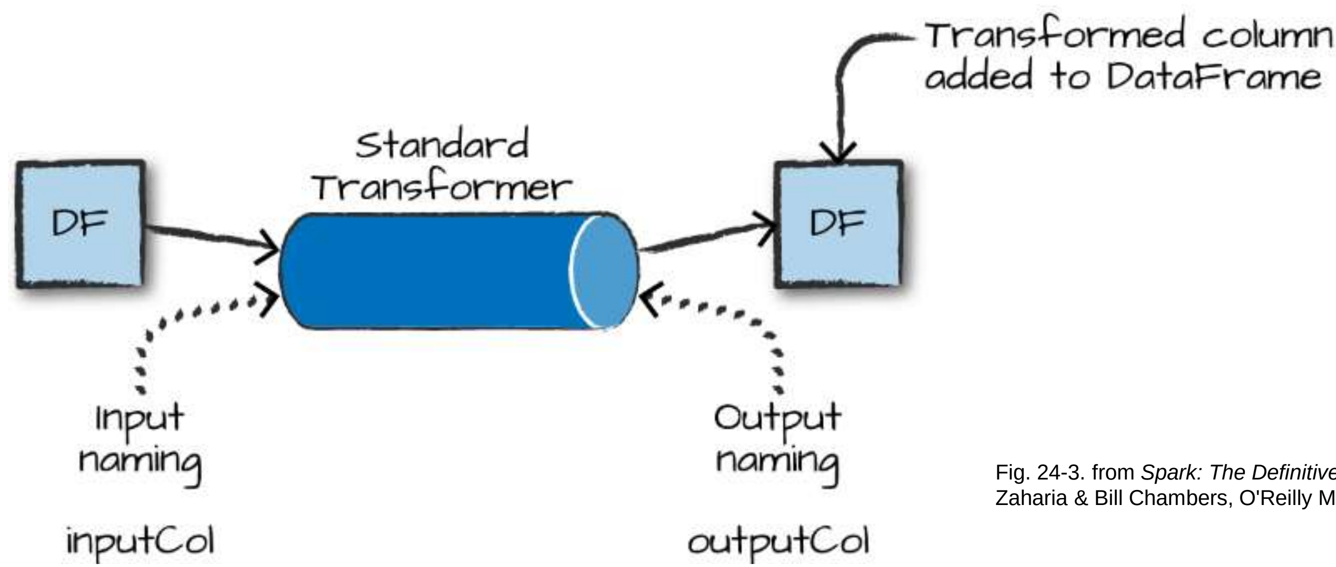


Fig. 24-3. from *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Estimators: Details

- An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data
- Technically, an E. implements a method **fit()**, which accepts a DataFrame and produces a *model* (Transformer)
  - E.g., a learning algorithm such as LogisticRegression is an Estimator, and calling **fit()** trains a LogisticRegressionModel, which is a Model and hence a Transformer

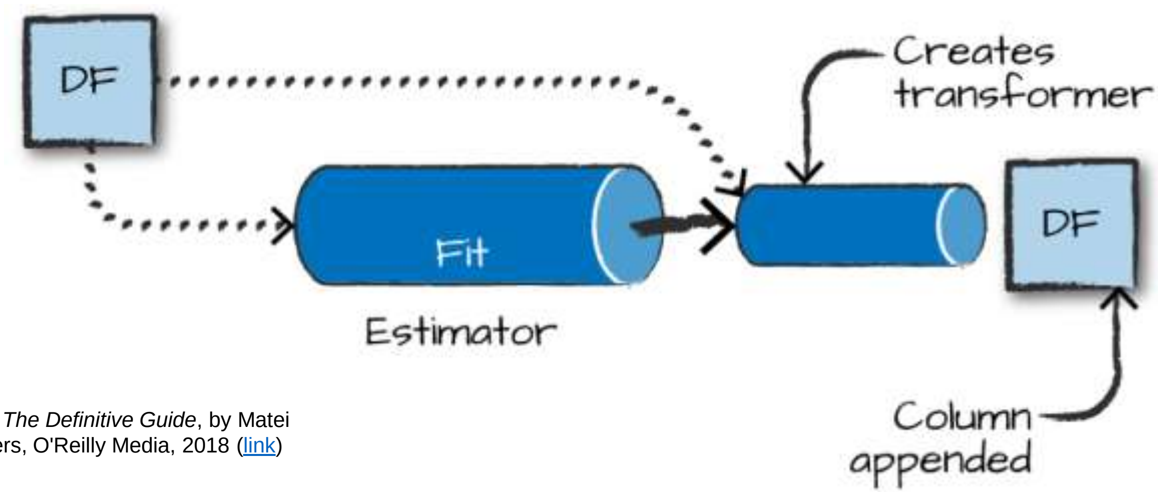


Fig. 25-2. from *Spark: The Definitive Guide*, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

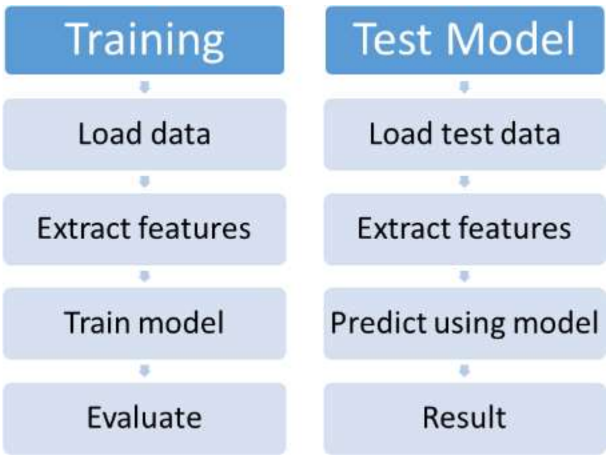
Pipelines as Sequences

- In ML it is common to run a sequence of algorithms to process and learn from data
- E.g., a simple text document processing workflow might include several stages:
  - Split each document's text into words
  - Convert each word into a numerical feature vector
  - Learn a prediction model using the feature vectors and labels
- MLlib represens this workflow as a **Pipeline**
  - It consists of a sequence of **PipelineStages** (Transformers and Estimators) to be run in a specific order

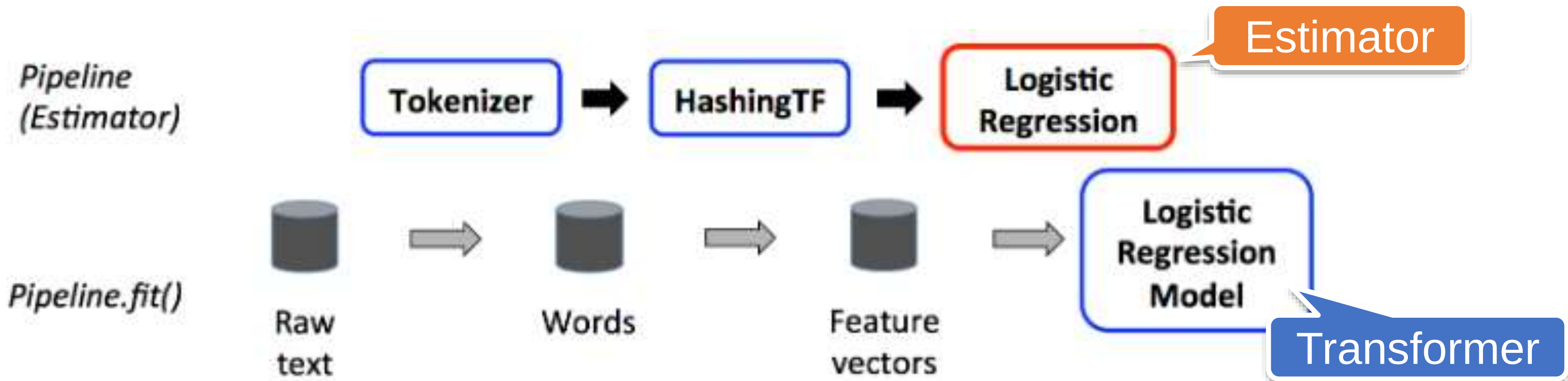
Pipelines vs. PipelineModels

- In an ML study, we typically have these subtasks:
  - Training: build model(s) according to data
  - Test/prediction: apply models to new data
- The ML Pipelines API supports both subtasks
- A complete "untrained" Pipeline ***pipe*** is an Estimator
- Training:
  - We call ***pipe***.fit()
  - The result is an object ***model*** of type **PipelineModel**, which is a Transformer
- Test/prediction:
  - Set test data as input to ***model***
  - Call ***model***.transform() to perform predictions on test data

Pipeline (Estimator) | PipelineModel (Transformer)



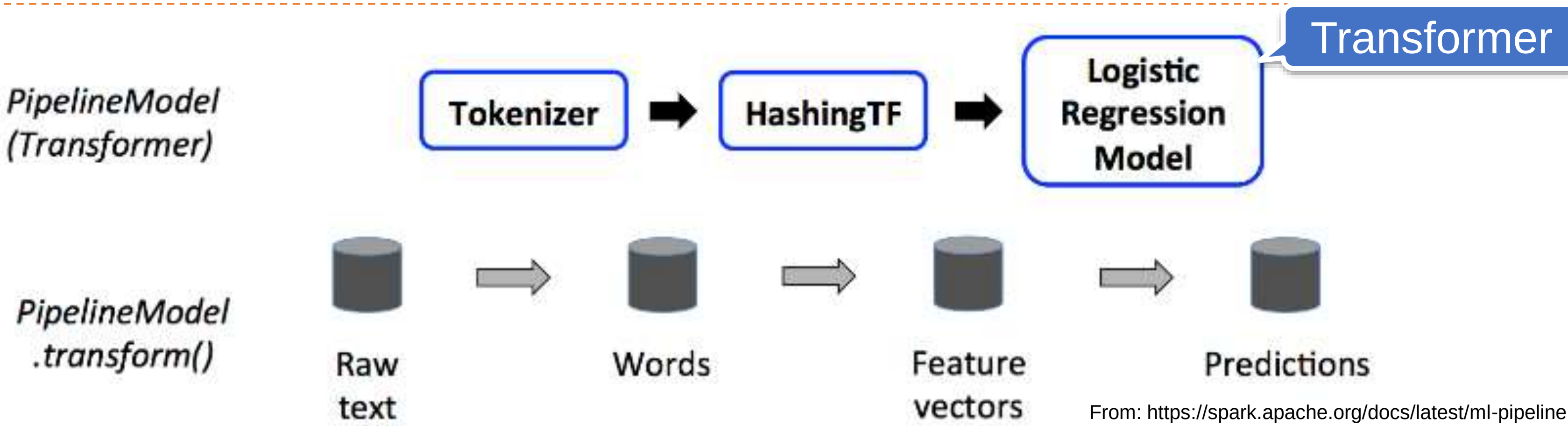
Example Pipeline Usage: Training Time



From: https://spark.apache.org/docs/latest/ml-pipeline.html

- The Pipeline.fit() method is called on the original DataFrame with raw text documents and labels
  - Recall: the *complete* pipeline is of type Estimator
- Pipeline calls the transform() methods of Tokenizer and HashingTF, then fit() method of Log. Regression
- After a Pipeline's fit() method runs, it produces a **PipelineModel**, which is a Transformer (=> phase 2)

Example Pipeline Usage: Prediction Time



From: https://spark.apache.org/docs/latest/ml-pipeline.html

- The PipelineModel has the same number of stages as the original Pipeline, but all Estimators become Transformers!
- When the PipelineModel's transform() method is called on a test/production dataset, the data are passed through the fitted (= trained) pipeline in order
- In particular, the LogististicRegressionModel performs classification (predictions) due to model trained in 1. phase

Vectors

- We need some lower-level data types, esp. **Vector**
- To pass a set of features to a model, we need to do it as a vector that consists of Doubles
- Vectors can be either sparse (where most of the elements are zero) or dense (“normal”)
  - Create sparse: specify an array of all the values
  - Create dense: specify the total size and the indices and values of the non-zero elements

```
from pyspark.ml.linalg import Vectors
denseVec = Vectors.dense(1.0, 2.0, 3.0)
size = 3
idx = [1, 2] # locations of non-zero elements in vector
values = [2.0, 3.0]
          sparseVec = Vectors.sparse(size, idx, values)
```

Example: Pipeline (Training) (link)

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Example: Pipeline (Prediction)

```
# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s,
            prediction=%f" % (rid, text, str(prob), prediction))
```

Implemented Functionality in MLlib /1

MLlib provides a large variety of scalable functions:

- Feature Engineering
  - Extraction: Extracting features from “raw” data
  - Transformation: Scaling, converting, or modifying features
  - Selection: Selecting a subset from a larger set of features
  - Locality Sensitive Hashing (LSH)
- Classification
  - Logistic regression, decision tree, random forest,
  - Gradient-boosted trees, multilayer perceptron, linear SVM
- Regression
  - (Generalized) linear regression, decision tree, random forest, .., survival regression, isotonic regression

Implemented Functionality in MLlib /2

- Clustering
  - K-means, Latent Dirichlet allocation (LDA), Bisecting k-means, Gaussian Mixture Model (GMM)
- Collaborative filtering (recommender systems)
- Frequent Pattern Mining
  - FP-Growth, PrefixSpan
- Model selection and hyperparameter tuning
  - Model selection using Pipelines (CrossValidator and TrainValidationSplit)

Overview Feature Engineering Functions

- Feature Extractors
  - TF-IDF
  - Word2Vec
  - CountVectorizer
  - FeatureHasher
- Feature Transformers
  - Tokenizer
  - StopWordsRemover
  - n-gram
  - Binarizer
  - PCA
  - PolynomialExpansion
  - Discrete Cosine Transform (DCT)
  - StringIndexer
  - IndexToString
  - OneHotEncoderEstimator
  - VectorIndexer
  - Interaction
  - Normalizer
  - StandardScaler
  - MinMaxScaler
  - MaxAbsScaler
  - Bucketizer
  - ElementwiseProduct
  - SQLTransformer
  - VectorAssembler
  - VectorSizeHint
  - QuantileDiscretizer
  - Imputer
- Feature Selectors
  - VectorSlicer
  - RFormula
  - ChiSqSelector
- Locality Sensitive Hashing
  - LSH Operations
  - LSH Algorithms

Examples Feature Engineering Functions

- Extractor: [Word2Vec]
  - Takes sequences of words and trains a Word2VecModel
  - The model maps each word to a unique fixed-size vector
- Feature Transformer: [QuantileDiscretizer]
  - Takes a column with continuous features and outputs a column with binned categorical features
  - The number of bins is set by the numBuckets parameter
- Feature Selector: [ChiSqSelector]
  - Uses the *Chi-Squared test of independence* to decide which features to choose
  - It supports 5 selection methods: numTopFeatures, percentile, fpr, fdr, few; e.g.
    - fpr chooses features whose *p-values* are below a threshold

More on Feature Engineering with Spark

Spark: The Definitive Guide, by Matei Zaharia & Bill Chambers, O'Reilly Media, 2018 (link)

Chapter 25. Preprocessing and Feature Engineering

- “Any data scientist worth her salt knows that one of the biggest challenges (and time sinks) in advanced analytics is preprocessing.
- It’s not that it’s particularly complicated programming, but rather that it requires deep knowledge of the data you are working with and an understanding of what your model needs in order to successfully leverage this data.”

Web Services: Overview

Web Services – User View

- **Web services** are extensions of the applications which are usable over HTTP
  - Extension of simple viewing and downloading web pages and other resources via generic browsers
  - Allow application-specific clients to interact with a service via a functionally-specialized interface over Internet

- When talking about WS, consider the **WS Lemma**:

- *For each statement* ***s*** *on Web Services and each* δ *with* $0 < \delta < 1$*, there is at least one famous guru* ***g*** *which contradicts s with probability* ***p*** $> \delta$.

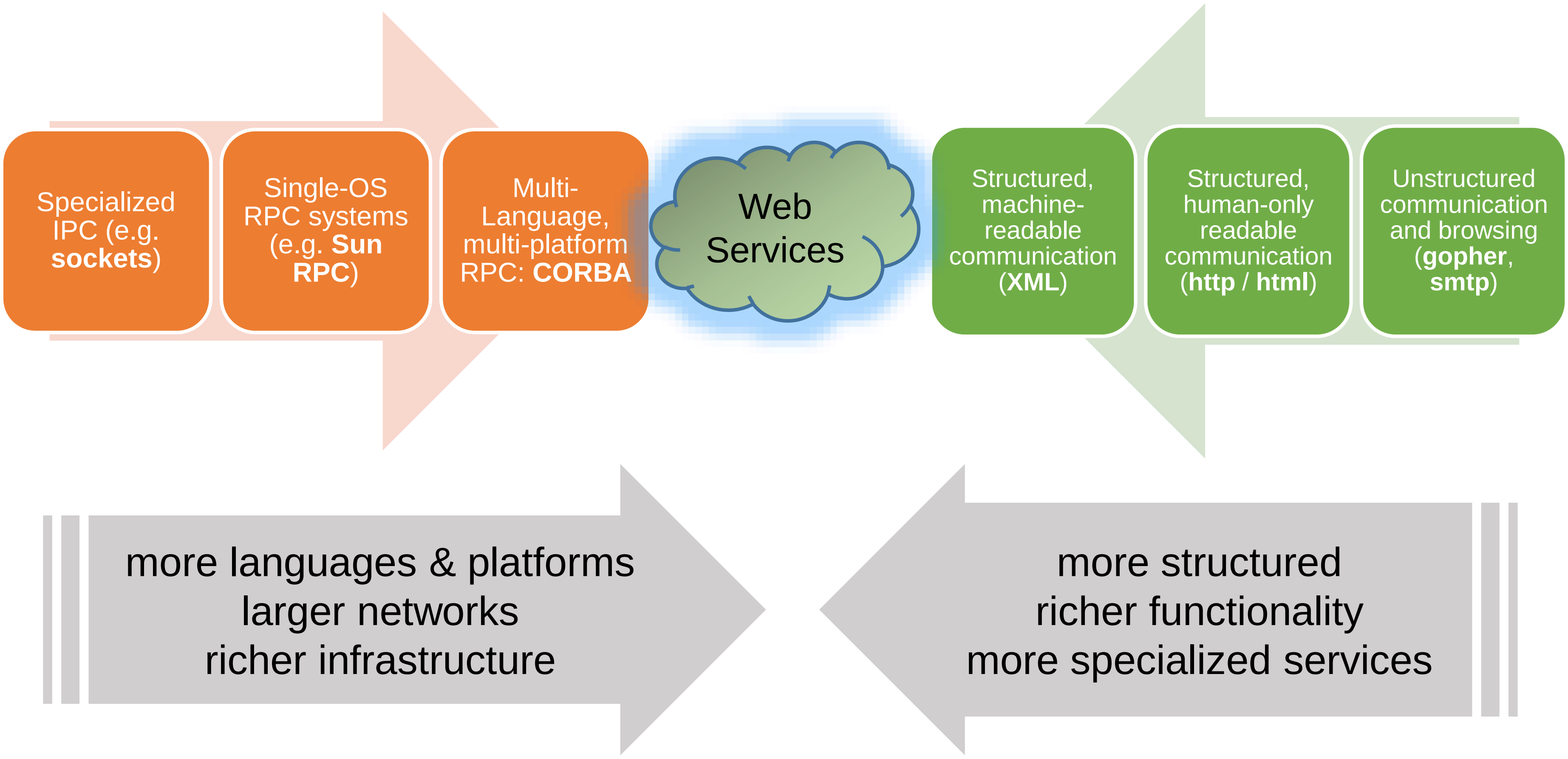
What are Web Services Exactly?

- Definition by W3C:
  - *A software system designed to support interoperable machine-to-machine interaction over a network*
- Frequently just APIs that can be accessed over a network and executed on a remote system
  - So essentially, special forms of RPC frameworks
- Why not CORBA, DCOM, Java RMI etc.?
  - These are good in local environments: high reliability, low latency, single management, no firewalls
  - Not suitable to call services of other owners / institutions over a potentially instable network connection
  - Further problems: firewalls prevent "standard" RPC calls

Evolution towards Web Services

Yet another definition of web services:

Systems where clients and servers communicate over HTTP



Commercial Web Service Examples

- Google:
  - Google Language API Family (link)
  - Google Maps JavaScript API V3 (link)
  - Google Analytics APIs (link)
  - Interesting: Google APIs Discovery Service (link)
  - For more, see http://code.google.com/apis
- Amazon - Amazon Web Services (AWS)
  - Amazon Elastic Compute Cloud (EC2) (link)
    - Virtual servers (cloud computing)
  - Amazon Simple Storage Service (Amazon S3) (link)
    - Cloud storage
  - Amazon Flexible Payments Service (FPS) (link)

Web Service Protocols (and Buzzwords)

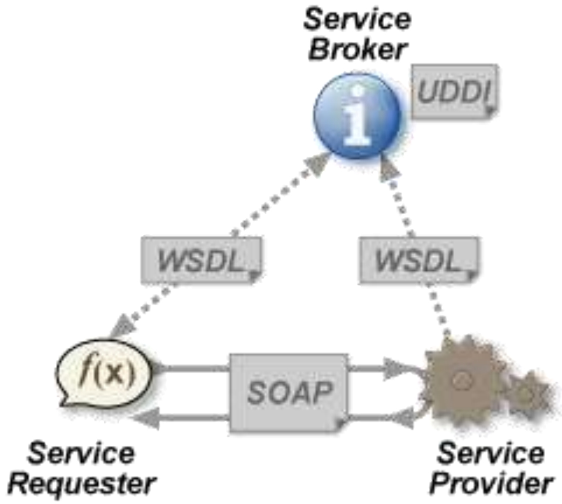
- **HTTP**: communications protocol
  - Hypertext Transfer Protocol
- **XML**: data format
  - eXtended Markup Language
- **SOAP**: format for requesting services
  - Simple Object Access Protocol
- **WSDL**: format for defining services
  - Web Services Definition Language
- **UDDI**: protocol for discovering services
  - Universal Description, Discovery, & Integration
- **REST**: Representational State Transfer
  - Software architecture for “web-like” web services

Two Types of Web Service Protocols

- "**Big**" (or **SOAP**) **Web Services**
  - Follow the SOAP-standard (later more on this)
  - More heavyweight, but provide arbitrary operations
- **REST**-compliant Web services
  - Primarily used to manipulate XML/JSON encodings of Web resources using stateless operations
- Note: We are talking only here only about **protocols** needed to communicate between remote components
  - The service functionality should be independent of these protocols
  - Protocols should be a minor aspect

"Big" (or SOAP) Web Services

- Use XML messages that follow the **SOAP** standard
  - SOAP: (originally) Simple Object Access Protocol
- Accompanied by a machine-readable description of the operations (methods) written in **WSDL**
  - WSDL = Web Services Description Language
  - Usually needed for automated client-side code generation in many Java and .NET SOAP frameworks
- W3C call them “arbitrary Web services”
  - As the service may expose an arbitrary set of operations
- Also called “RPC-Style” services



RESTful
Web Services

REST-compliant Web services

- **REST** means *Representational State Transfer*
  - Such services are also called **RESTful** Web Services
- Focus is on manipulating textual representations of web resources rather than messages or operations
- **Web resources**
  - … are typically documents or files identified by URLs
  - But can be any "entities" or pieces of data
  - Documents/entities are encoded in XML, HTML, or JSON
- General schema of RESTful services:
  - Requests specify resource's URI with parameters
  - Responses can provide links to other related resources, or confirm changes of the target resource

REST Protocols and Philosophy

- REST uses as a basis standard HTTP protocol with operations: GET, POST, PUT, DELETE
  - … or other **CRUD** HTTP methods
  - CRUD = create, read, update, and delete - the four basic functions of persistent storage
- REST uses a stateless protocol (HTTP) and standard operations – why?
- To achieve performance, reliability, and extensibility
  - Via re-using already existing (standard) components
  - Extensibility: components can be replaced even when the system is running

REST - Principles

- Application state and functionality are abstracted into resources
- Every resource is uniquely addressable using a universal syntax (URLs)
- All resources share a uniform interface for transfer of state between client and resource, consisting of
  - A constrained set of well-defined operations
    - In practice, these are HTTP-Operations
  - A constrained set of content types
- Protocol is: client-server, stateless, cacheable

REST - Implementation

- REST unifies the inter-system interfaces to small set of standardized actions (= verbs)
- Actions are essentially the same as in HTTP
  - GET requests data from the server (by the client)
  - POST stores new data/resources on the server
  - PUT updates existing data or complements subordinated resources
  - DELETE deletes the Data of the Client on the server
  - HEAD requests meta-data of a resource from the server
  - OPTIONS allow the client to see, which methods are available to a resource

Example: Dealer of Machine Parts

- Get parts info:

  HTTP GET //www.parts-depot.com/parts

- Returns a document containing a list of parts

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
         xmlns:xlink="http://www.w3.org/1999/xlink">
      <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
      <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
      <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
      <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

Example: Dealer of Machine Parts

- Get detailed info on a specific part:

  HTTP GET //www.parts-depot.com/parts/00345
- Returns a document containing a list of parts (implementation transparent to clients)


```
?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
        xmlns:xlink="http://www.w3.org/1999/xlink">
    <Part-ID>00345</Part-ID>
    <Name>Widget-A</Name>
    <Description>This part is used within the frap assembly</Description>
    <Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
    <UnitCost currency="USD">0.10</UnitCost>
    <Quantity>10</Quantity>
</p:Part>
```

RPC vs. REST (Example from Wikipedia)

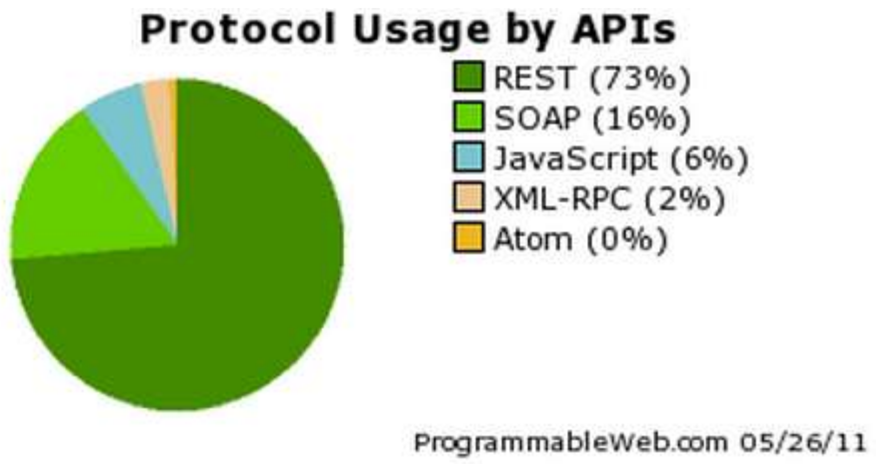
- RPC-style methods:
  - getUser(), addUser(), removeUser(), updateUser(), getLocation(), AddLocation(), removeLocation()
  - Each needs own “syntax” in SOAP
  - Binding to client library:
    - exampleObject = new ExampleApp(“example.com:1234”);
    - exampleObject.getUser();
- REST-style “methods”:
  - http://example.com/users
  - http://example.com/users/{user}
  - http://example.com/locations
  - Binding to client library

    userResource = new Resource(“http://example.com/users/001”);

    userResource.get();

REST Takes Over

- http://www.programmableweb.com/apis



- Why?
  - Maybe: in long term, simplicity wins
  - See presentation by Willy Zwaenepoel (EPFL): *P2P, DSM, and Other Products of the Complexity Factory* (PPT via Google search)

Outlook:
gRPC and Protocol Buffers

Simplicity Wins Again

- Trend: almost all web services use only JSON as encoding and no XML (see [link])
  - **JSON**: *JavaScript Object Notation*
- JSON uses *textual* representation: can we go **binary**?
- Meet **gRPC** and **protocol buffers** (proto buffers)
- **gRPC** (Google RPC) - https://grpc.io/
  - A high performance, open-source RPC framework
  - gRPC uses for serialization …
- **(Google) Protocol Buffers** - link
  - A language/platform-neutral mechanism for serializing structured data

gRPC and Proto Buffers

- gRPC features
  - A (i) standard, (ii) libraries, and (iii) tools for code generation
  - Works across languages and platforms
  - Scale to millions of RPCs per second
  - Bi-directional streaming and integrated auth
  - Bindings for C++, Java, Python, Dart, Go, Ruby, C#, Objective C, JavaScript, PHP, Node.js, and some more
- Proto Buffers
  - Idea: transform binary data from/to OOP-code easily
  - A (i) format description (.proto – files), (ii) libraries, and (iii) tools for code generation
  - Protocol buffers are 3 to 10 times faster than JSON serialization, and 20 to 100 times faster than XML
  - Simple but versatile message definition standard

Example gRPC Message Format (Link)

```
// The greeting service definition
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Real gRPC Definition Example

```
// Artur Andrzejak
// September 2018
syntax = "proto3";
option java_multiple_files = true;
option java_package = "recommender_rpc";
option java_outer_classname = "RecommenderRpcProto";
option objc_class_prefix = "RcmdrRPC";

package recommender_rpc;
service RecommenderService {
    rpc InitSession (InitialisationReq) returns (InitialisationResp) {}
    rpc CloseSession (DisposeRecommenderReq) returns (DisposeRecommenderResp) {}
    rpc onUserInput (OnNewInputReq) returns (SuggestionsSetResp) {}
    rpc suggestionSelected (OnSuggestionSelectedReq) returns (SuggestionDetailsResp) {}
}

// The initialisation of the recommender service
message InitialisationReq {
    // Path to a file with settings of the recommender
    string settings_path = 1;
    // Targeted prg language (python, java, ..)
    string prog_language = 2;
    // Path to the edited source file
    string src_file_path = 3;
    // Path to all sources
    string src_dir_path = 4;
    // Optional labels for domains of the developed code (e.g. pandas, numpy, ..)
    repeated string domain_labels = 5;
}
// The response message containing the sessionId
message InitialisationResp {
    int64 sessionId = 1;
    string status = 2;  // OK or error message
}
```

```
message UserFocusHint {
    string name = 1;    // name of the variable, domain, keyword
    string value = 2;   // optional, for some cases obsolete
    enum HintType {
        VARIABLE = 0;
        DOMAIN = 1;
        KEYWORD = 2;
        SETTING = 3;
    }
    HintType hintType = 3;
    enum HintImpact {
        INCLUDE_ALLWAYS = 0;    // means: always include this var in suggested fragments
        PREFER = 1;             // means: prefer this var in suggested fragments
        EXCLUDE_ALLWAYS = 2;    // means: always exclude this var from suggested fragments
    }
}

message OnNewInputReq {
    int64 sessionId = 1;
    string searchText = 2;
    enum ChangeType {
        SEARCHTEXT = 0;         // user changed the typed search text
        VARVIEWERSTATE = 1;     // changes in the variable viewer state
        SOURCEFILE = 2;         // source file was changed
        PREFERENCES = 3;        // prefered vars, domains etc. changed
        OTHER = 10;       // should force complete re-computation
    }
    ChangeType changeType = 3;
    repeated Parameter activeVariables = 10;
    ViewerSelectionState viewerState = 20;
    repeated UserFocusHint userFocusHint= 30;
}


// A single code suggestion (i.e. suggested fragment)
```

Protocol Buffer Example - **.proto** File

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0; HOME = 1; WORK = 2; }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

Data Access Classes

- A protocol buffer compiler for a language generate <u>data access classes</u>
  - Accessors for each field
  - Methods to serialize/parse the whole structure to/from raw bytes

Sending data to a stream

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email
        ("jdoe@example.com");
fstream output("myfile", ios::out |
        ios::binary);
person.SerializeToOstream
        (&output);
```

Retrieving from a stream

```
fstream input("myfile", ios::in |
        ios::binary);
Person person;
person.ParseFromIstream
        (&input);
cout << "Name: " <<
        person.name() << endl;
cout << "E-mail: " <<
        person.email() << endl;
```

Thank you.

Additional Slides

XML-RPC and SOAP

XML-RPC

- A remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism
- Created 1998 by Dave Winer of UserLand Software and Microsoft
  - Later evolved into SOAP
- Data marshaled into XML messages
  - All request and responses are human-readable XML
- Explicit typing
- No true IDL compiler support
  - Lots of support libraries
- JSON-RPC (2005+) is similar to XML-RPC

XML-RPC example

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
     <param><value><int>40</int></value></param>
  </params>
</methodCall>
```

```
<?xml version="1.0"?>
<methodResponse>
    <params>
        <param>
            <value><string>South Dakota</string></value>
        </param>
    </params>
</methodResponse>
```

XML-RPC data types

- int
- string
- boolean
- double
- dateTime.iso8601
- base64
- array
- struct

Examples Data Types

Array

- **<array>**
  - **<data>**
    - **<value><i4>**1404**</i4></value>**
    - **<value><string>**Something here**</string></value>**
      <value><i4>1**</i4></value>**
  - **</data>**
- **</array>**

Struct

- **<struct>**
  - **<member>**
    - **<name>**foo**</name>**
    - **<value><i4>**1**</i4></value>**
  - **</member>**
  - **<member> … </member>**
- **</struct>**

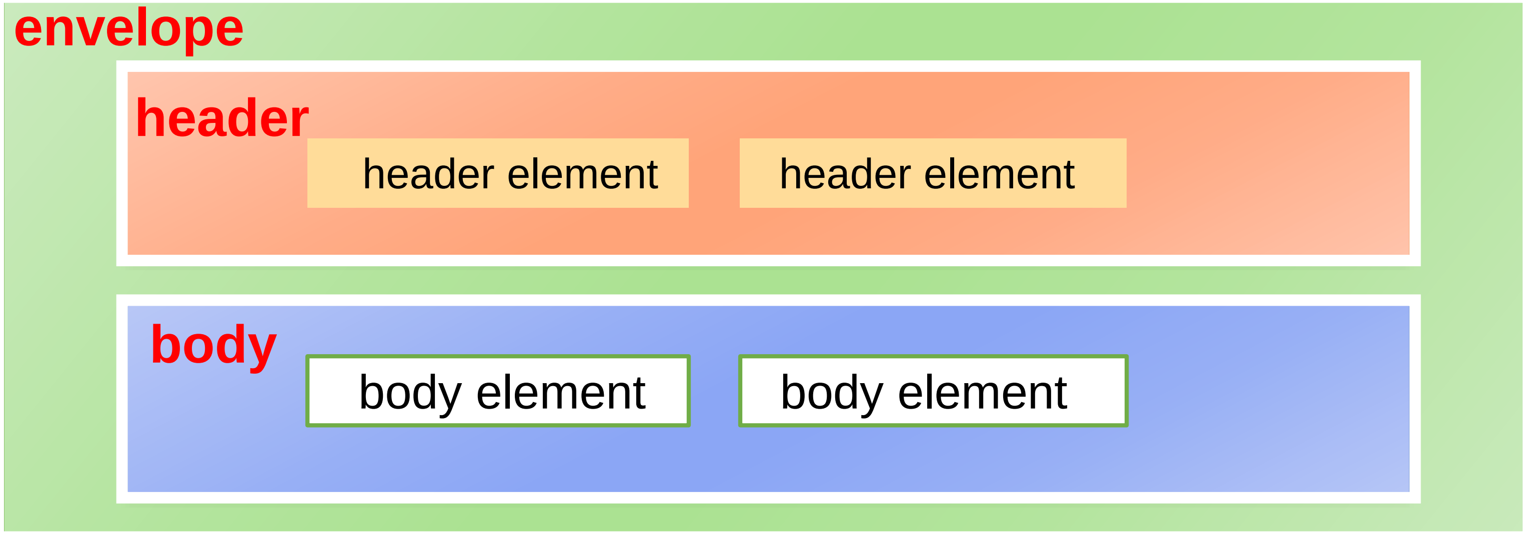
XML-RPC Assessment

- Simple (spec about 7 pages) and humble goals
- Good language support
- Little/no industry support
  - Mostly developed by users
- XML-RPC is simpler to use than SOAP because it
  - allows only one method of method serialization
    - SOAP defines multiple different encodings
  - has a simpler security model
  - does not require/support) service descriptions in WSDL
    - Yet XRDL provides similar functionality as WSDL

SOAP - Simple Object Access Protocol

- 1998 and evolving (v1.2 Jan 2003)
  - A W3C recommendation since June 24, 2003
- Specifies rules for using XML to package messages
  - Not necessarily for RPC
- Continues where XML-RPC left off
  - user defined data types
  - ability to specify the recipient
  - message specific processing control
- XML (usually) over HTTP
  - But currently also over SMTP, TCP or UDP

SOAP Message



- Header is optional
  - Used for establishing the necessary context for a service or for keeping a log or audit of operations
- Body contains the message (e.g. service request)

Example Message

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
```

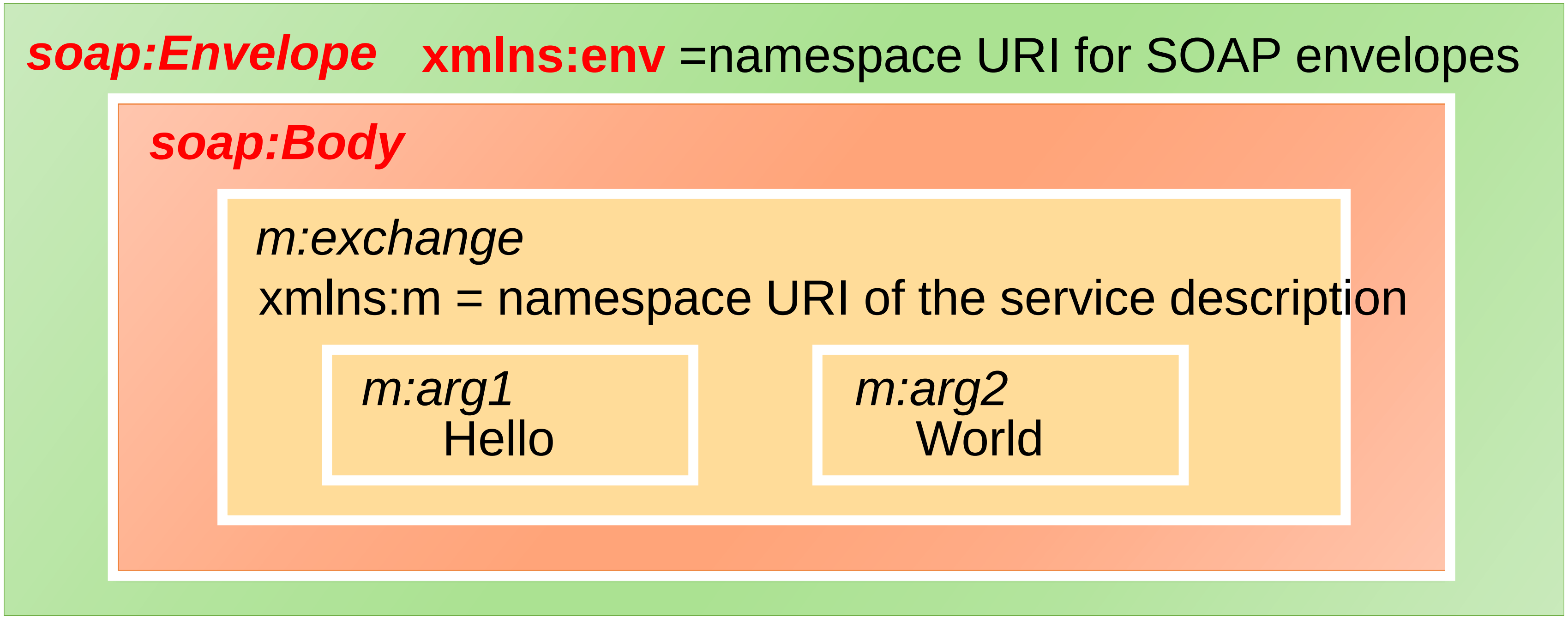
What is call method name and parameter?

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap =
  "http://www.w3.org/2003/05/soap-envelope">
  <soap:Header> </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Method name

Method actual parameter

Generic SOAP Message Format



- Body: arbitrary XML; only requirements
  - The **soap:Envelope** and **soap:Body** elements must be present
  - soap:Envelope character encoding must be the same as in the enclosing element

Each XML element is represented by a shaded box with its name in italic followed by any attributes and its content

SOAP over HTTP Uses POST for a Request

```
POST /examples/stringer
Host: www.cdk4.net
Content-Type: application/soap+xml
Action: http://www.cdk4.net/examples/stringer#exchange

<?xml version="1.0"?>
<soap:Envelope xmlns:soap = "http://www.w3.org/2003/05/soap-envelope">
	<soap:Header> ...  </soap:Header>
	<soap:Body>
		...
	</soap:Body>
</soap:Envelope>
```

endpoint address

action

HTTP header

Soap message

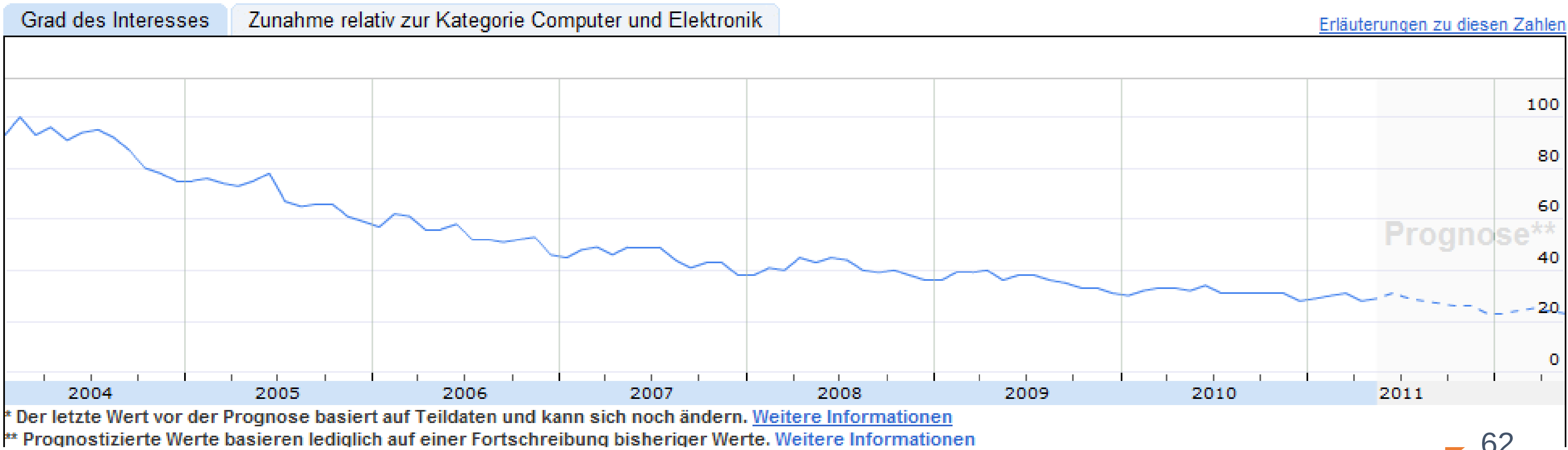
- HTTP headers specify the endpoint address (URI of the ultimate server) and the action to be carried out
  - Action parameter optimizes dispatching – it reveals the operation without the need to analyze the SOAP message
- HTTP body carries the messages

SOAP API Example

- Google Web Search API - deprecated since in 2006
- Still, useful resources to download
  - A WSDL file you can use with any development platform that supports web services
  - A Java library that provides a wrapper around the Google Web APIs SOAP interface
  - An example .NET program which invokes the Google Web APIs service
  - Documentation that describes the SOAP API and the Java library

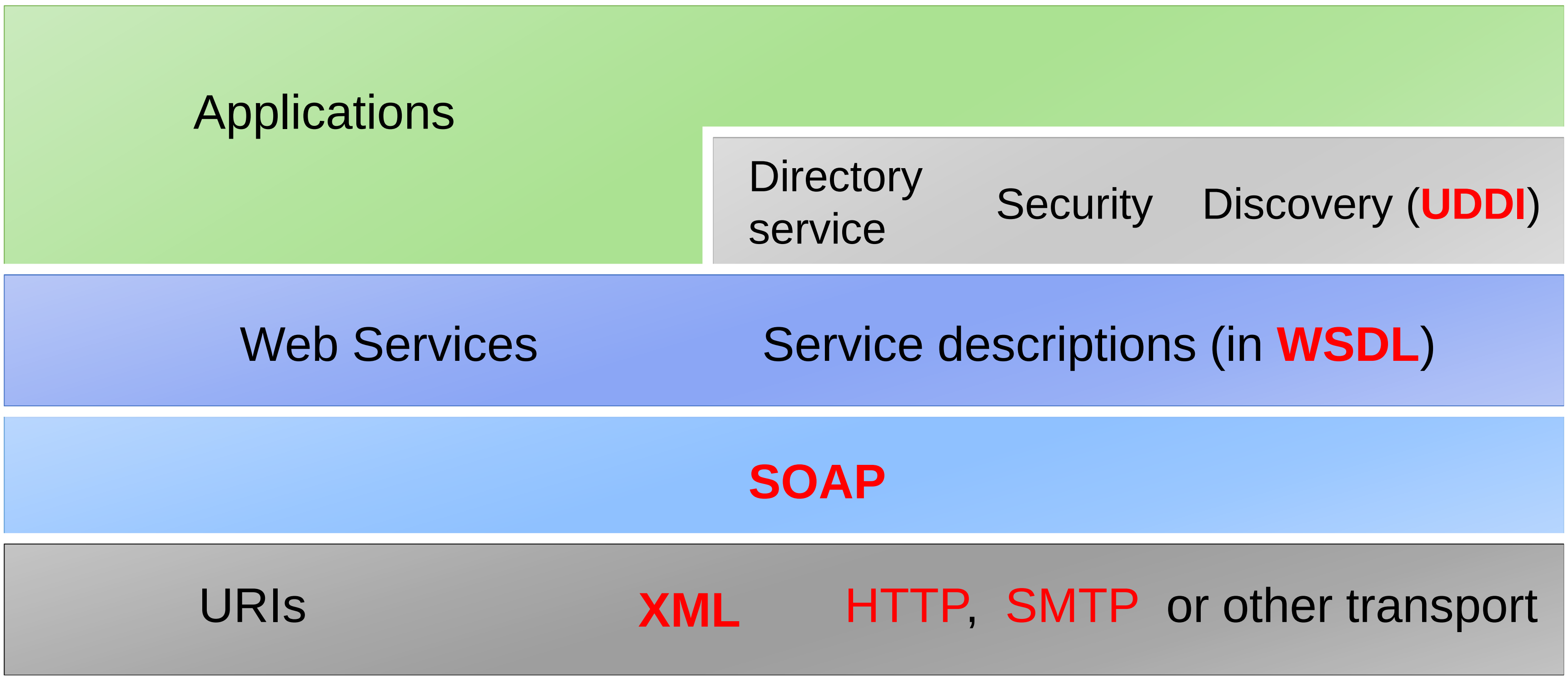
The Future of SOAP?

- Many Amazon WSs offer REST and SOAP APIs
- Allegedly complex because “*we want our tools to read it, not people*” (unnamed Microsoft employee)
- Google Insights for Search
  - http://www.google.com/insights/search/#cat=0-5&q=soap&cmpt=q



WSDL

“Big Web Service” Protocols Stack



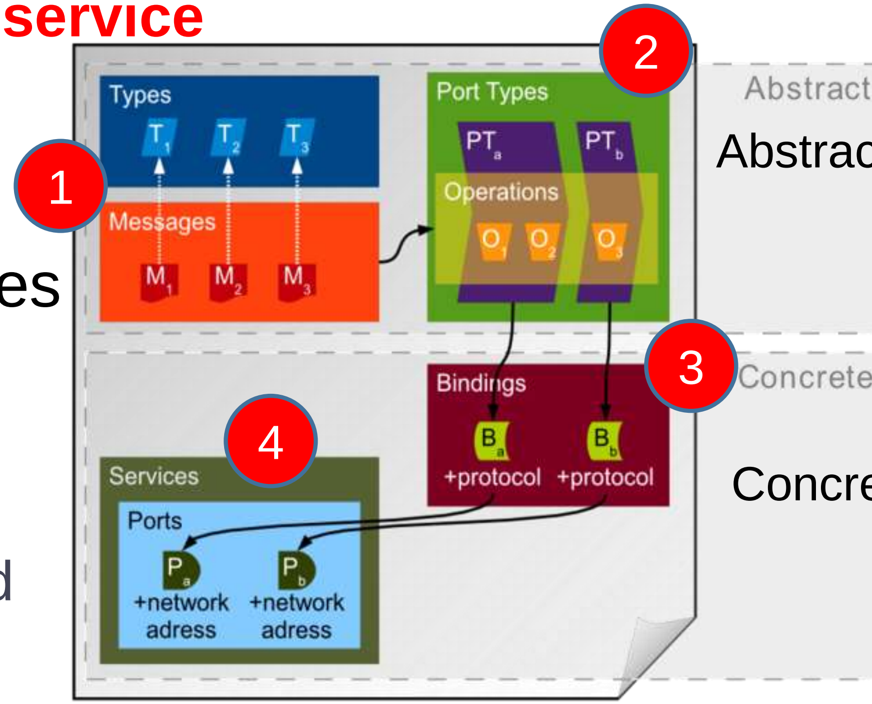
Web Services and WSDL

- Web Services Description Language (WSDL) describes the services
  - interfaces and information such as server‘s URL
  - Analogous to an IDL
  - Describe capabilities of an organization’s web services
- Current standard is WSDL 2.0
  - Endorsed by W3C in June 2007
  - Version 2.0 has bindings (API equivalents) to all HTTP request methods
    - 1.1 supported only GET and POST
  - 2.0 is much simpler to implement

WSDL 1.1 Key Concepts

1. **Messages** are abstract descriptions of the data being exchanged (and have own **types**)
2. **Port types** (interfaces) define the “syntax” of **operations**
3. **Port** is is a “service address”, defined by associating a network address with a reusable **binding**
4. A collection of ports define a **service**

- <u>Concrete</u> protocol and data format specifications for a particular port type constitutes a reusable binding
  - here the operations and messages are bound to a concrete network protocol and message format

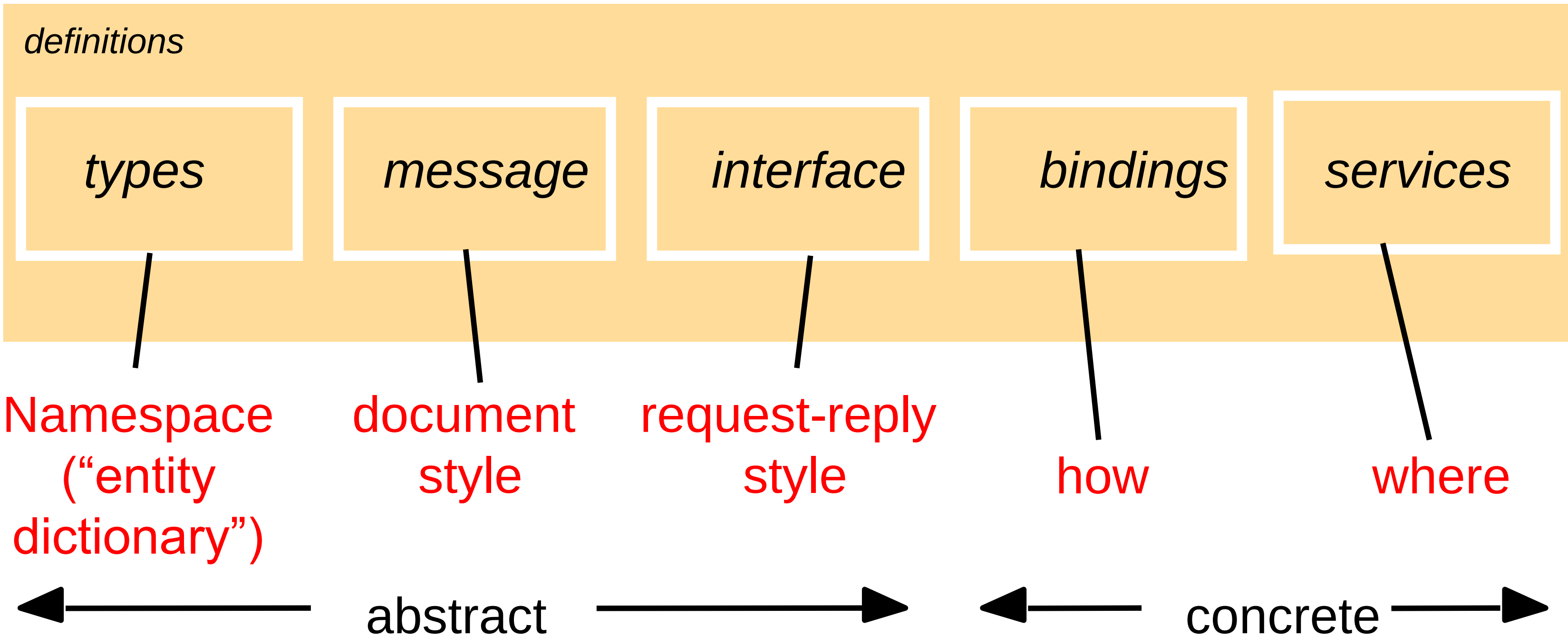


Abstract

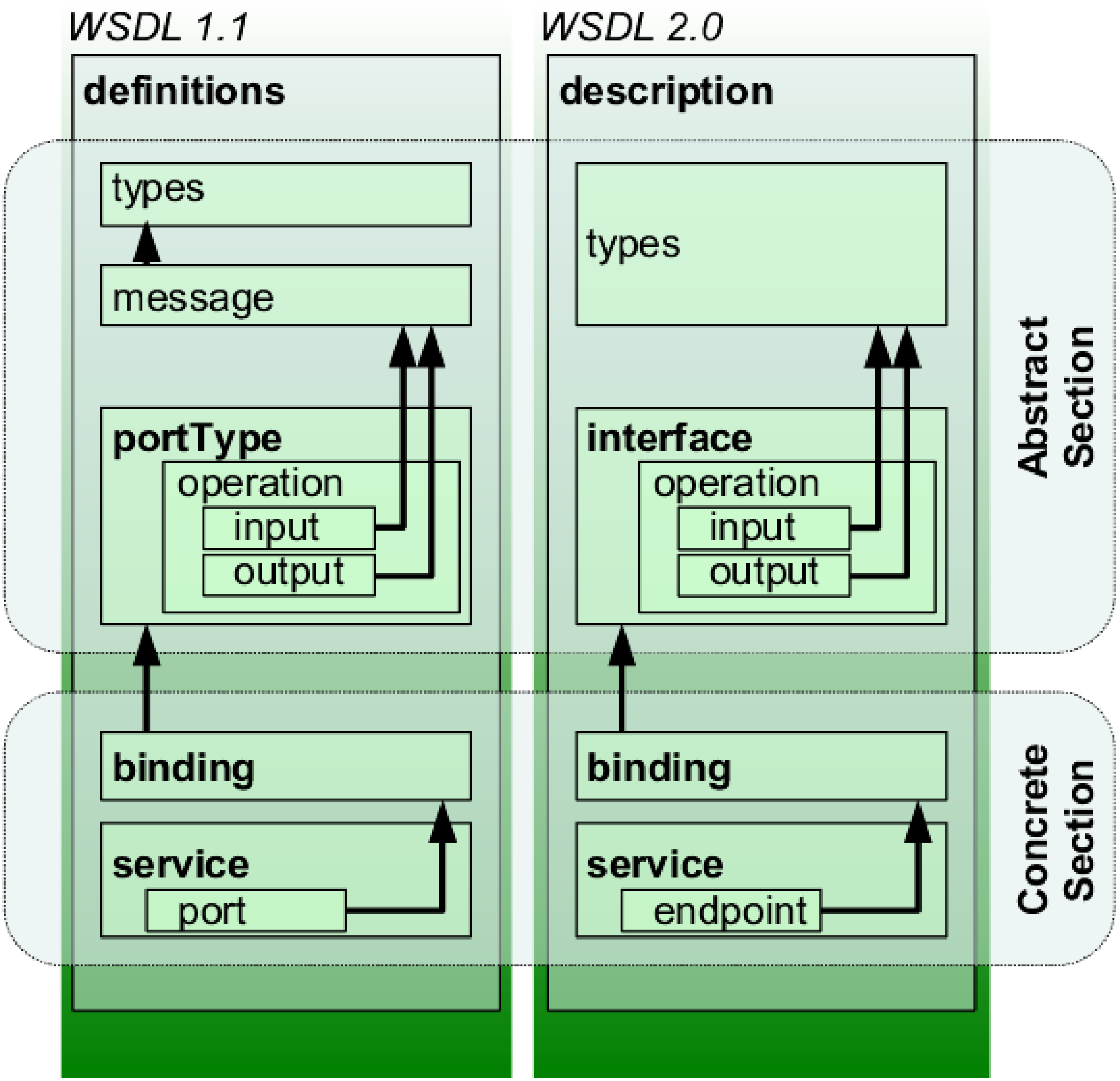
Concrete

Source: Wikipedia

WSDL 1.1 Document Structure



Comparison WSDL 1.1 vs. WSDL 2.0



Source: [Wikipedia]()

Comparison WSDL 1.1 vs. WSDL 2.0

- Service
  - Container for a set of functions exposed to external use
- Port / Endpoint
  - Defines the address or connection point to a Web service
  - Typically represented by a simple HTTP URL
- Binding / Binding
  - Defines the operations and the interface
  - Other specifications, e.g. transport type (SOAP Protocol), the SOAP binding style (RPC/Document)
- PortType / Interface
  - Element <portType> or <interface> in WSDL 2.0
    - Defines a web service, the operations that can be performed, and the messages which can be used

Comparison WSDL 1.1 vs. WSDL 2.0

- Operation
  - Essentially, a method or function call
  - In addition to the action specifies the way how message is encoded, for example, "literal”
- Message / N.A.
  - Contains the information needed to perform a single operation
  - Removed in WSDL 2.0; instead, XML schema types are used for defining bodies of inputs, outputs and faults

WSDL Example – Abstract Types

```
<types> <!-- Abstract types -->
  <xs:schema
 xmlns="http://www.example.com/wsdl20sample"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.example.com/wsdl20sample">
    <xs:element name="request">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="header"
 maxOccurs="unbounded">
            <xs:complexType>
              <xs:simpleContent>
                <xs:extension base="xs:string">
                  <xs:attribute name="name"
 type="xs:string" use="required"/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
          <xs:element
 name="body" type="xs:anyType"
 minOccurs="0"/>
        </xs:sequence>
        <xs:attribute
 name="method" type="xs:string"
 use="required"/>
        <xs:attribute name="uri"
 type="xs:anyURI" use="required"/>
      </xs:complexType>
    </xs:element>
```

This example will be replaced be
http://weblogs.com/ ping-service WSDL!

WSDL Example –Interfaces

```
<!-- Abstract interfaces -->
  <interface name="RESTfullInterface">
    <fault name="ClientError" element="tns:response"/>
    <fault name="ServerError" element="tns:response"/>
    <fault name="Redirection" element="tns:response"/>

   <operation name="Get" pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="GetMsg"        element="tns:request"/>
      <output messageLabel="SuccessfulMsg" element="tns:response"/>
    </operation>

  <operation name="Post" pattern="http://www.w3.org/ns/wsdl/in-out">
      <input messageLabel="PostMsg"       element="tns:request"/>
      <output messageLabel="SuccessfulMsg" element="tns:response"/>
    </operation>
   ....
  </interface>
```

Concrete Binding over HTTP

```
<binding name="RESTfullInterfaceHttpBinding"
        interface="tns:RESTfullInterface"
         type="http://www.w3.org/ns/wsdl/http">
    <operation ref="tns:Get" whttp:method="GET"/>
    <operation ref="tns:Post" whttp:method="POST"
            whttp:inputSerialization= "application/x-www-form-urlencoded"/>
    <operation ref="tns:Put" whttp:method="PUT"
            whttp:inputSerialization= "application/x-www-form-urlencoded"/>
    <operation ref="tns:Delete"
  whttp:method="DELETE"/>
  </binding>
```

Concrete Binding with SOAP

```
<binding
  name="RESTfullInterfaceSoapBinding"
  interface = "tns:RESTfullInterface"
  type = "http://www.w3.org/ns/wsdl/soap"
  wsoap:protocol = "http://www.w3.org/2003/05/soap/bindings/HTTP/"
  wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
    <operation ref="tns:Get" />
    <operation ref="tns:Post" />
    <operation ref="tns:Put" />
    <operation ref="tns:Delete" />
</binding>
```

WSDL Example - Service

```
<service
   name="RESTfulService"
   interface="tns:RESTfullInterface">
   <endpoint name="RESTfulServiceHttpEndpoint”
          binding="tns:RESTfullInterfaceHttpBinding”
          address="http://www.example.com/rest/"/>
  <endpoint name="RESTfulServiceSoapEndpoint”
          binding="tns:RESTfullInterfaceSoapBinding“
          address="http://www.example.com/soap/"/>
</service>
```