

Informatische Werkzeuge in den Geistes- und Sozialwissenschaften 2

Prof. Dr. Michael Kohlhase
Professur für Wissensrepräsentation und -verarbeitung
Informatik, FAU Erlangen-Nürnberg
`Michael.Kohlhase@FAU.de`

2024-02-08

This document contains the course notes of the course “Informatische Werkzeuge in den Geistes- und Sozialwissenschaften IWGS-2” held at FAU Erlangen-Nürnberg in the Summer Semesters 2019 ff. Other parts of the lecture notes can be found at http://kwarc.info/teaching/IWGS/notes-*.pdf.

Contents

8 Semester Change-Over	1
8.1 Administrativa	1
9 Databases	9
9.1 Introduction	9
9.2 Relational Databases	11
9.3 SQL – A Standardized Interface to RDBMS	13
9.4 ER-Diagrams and Complex Database Schemata	16
9.5 RDBMS in Python	19
9.6 Excursion: Programming with Exceptions in Python	21
9.7 Querying and Views in SQL	23
9.8 Querying via Python	26
9.9 Real-Life Input/Output: XML and JSON	29
9.10 Exercises	35
10 Project: A Web GUI for a Books Database	39
10.1 A Basic Web Application	39
10.2 Access Control and Management	48
10.3 Asynchronous Loading in Modern Web Apps	52
10.4 Deploying the Books Application as a Program	60
11 Image Processing	63
11.1 Basics of Image Processing	63
11.1.1 Image Representations	63
11.1.2 Basic Image Processing in Python	70
11.1.3 Edge Detection	74
11.1.4 Scalable Vector Graphics	77
11.2 Project: An Image Annotation Tool	82
11.3 Fun with Image Operations: CSS Filters	89
11.4 Exercises	93
12 Ontologies, Semantic Web for Cultural Heritage	97
12.1 Documenting our Cultural Heritage	97
12.2 Systems for Documenting the Cultural Heritage	100
12.3 The Semantic Web	104
12.4 Semantic Networks and Ontologies	109
12.5 CIDOC CRM: An Ontology for Cultural Heritage	114
12.6 The Semantic Web Technology Stack	120
12.7 Ontologies vs. Databases	127
12.8 Exercises	130

13 The WissKI System	131
13.1 WissKI extends Drupal	131
13.2 Dealing with Ontology Paths: The WissKI Pathbuilder	135
13.3 The WissKI Link Block	139
13.4 Cultural Heritage Research: Querying WissKI Resources	141
13.5 Application Ontologies in WissKI	143
13.6 The Linked Open Data Cloud	145

Recorded Syllabus – Summer 2024: The recorded syllabus for this semester is in the course page in the ALEA system at <https://courses.voll-ki.fau.de/course-home/iwgs-2>. The table of contents in the IWGS notes at <https://courses.voll-ki.fau.de> indicates the material covered to date in yellow.

Chapter 8



Semester Change-Over

8.1 Administrativa

We will now go through the ground rules for the course. This is a kind of a social contract between the instructor and the students. Both have to keep their side of the deal to make learning as **efficient** and painless as possible.

Prerequisites

- ▷ **Formal Prerequisite:** IWGS-1 (If you did not take it, read the notes)
- ▷ **General Prerequisites:** Motivation, interest, curiosity, hard work.
nothing else! (apart from IWGS-1)
We will teach you all you need to know
- ▷ You can do this course if you want! (we will help)

 FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge © G/SW 2 206 2024-02-08 

Now we come to a topic that is always interesting to the students: the grading scheme: The short story is that things are complicated. We have to strike a good balance between what is didactically useful and what is allowed by Bavarian law and the FAU rules.

Assessment, Grades

- ▷ **Grading Background/Theory:** Only modules are graded! (by the law)
 - ▷ Module “DH-Einführung” (DHE) $\hat{=}$ courses IWGS1/2, DH-Einführung.
 - ▷ DHE module grade \rightsquigarrow pass/fail determined by “portfolio” $\hat{=}$ collection of contributions/assessments.
- ▷ **Assessment Practice:** The IWGS assessments in the “portfolio” consist of
 - ▷ weekly homework assignments, (practice IWGS concepts and tools)
 - ▷ 60 minutes exam directly after lectures end: July 27. 2024.
- ▷ **Retake Exam:** 60 min exam at the end of the exam break. (October. 12. 2024)

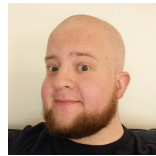
Homework assignments, and end-semester exam may seem like a lot of work – and indeed they are – but you will need practice (getting your hands dirty) to master the concepts. We will go into the details next.

IWGS Homework Assignments

- ▷ **Homeworks:** will be small individual problem/programming/system assignments
 - ▷ but take time to solve (at least read them directly ~> questions)
 - ▷ group submission if and only if explicitly permitted.
- ▷ ⚠ Without trying the homework assignments you are unlikely to pass the exam.
- ▷ **Admin:** To keep things running smoothly
 - ▷ Homeworks will be posted on StudOn.
 - ▷ Sign up for IWGS under <https://www.studon.fau.de/frm5075965.html>.
 - ▷ Homeworks are handed in electronically there. (plain text, program files, PDF)
 - ▷ Go to the tutorials, discuss with your TA! (they are there for you!)
- ▷ **Homework Discipline:**
 - ▷ Start early! (many assignments need more than one evening's work)
 - ▷ Don't start by sitting at a blank screen (talking & study group help)
 - ▷ Humans will be trying to understand the text/code/math when grading it.

It is very well-established experience that without doing the homework assignments (or something similar) on your own, you will not master the concepts, you will not even be able to ask sensible questions, and take nothing home from the course. Just sitting in the course and nodding is not enough! If you have questions please make sure you discuss them with the instructor, the teaching assistants, or your fellow students. There are three sensible venues for such discussions: online in the lecture, in the tutorials, which we discuss now, or in the course forum – see below. Finally, it is always a very good idea to form study groups with your friends.

IWGS Tutorials

- ▷ Weekly tutorials and homework assignments (first one in week two)
 - Tutor:** (Doctoral Student in CS)
 - ▷ Jonas Betzendahl: jonas.betzendahl@fau.de 
 - ▷ They know what they are doing and really want to help you learn! (dedicated to DH)
- ▷ **Goal 1:** Reinforce what was taught in class (important pillar of the IWGS concept)
- ▷ **Goal 2:** Let you experiment with Python (think of them as Programming Labs)

- ▷ **Life-saving Advice:** go to your tutorial, and prepare it by having looked at the slides and the homework assignments
- ▷ **Inverted Classroom:** the latest craze in didactics (works well if done right)
in IWGS: Lecture + Homework assignments + Tutorials $\hat{=}$ inverted classroom

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 209 2024-02-08

Do use the opportunity to discuss the IWGS topics with others. After all, one of the non-trivial inter/transdisciplinary skills you want to learn in the course is how to talk about computer science topics – maybe even with real computer scientists. And that takes practice, practice, and practice.

But what if you are not in a lecture or tutorial and want to find out more about the IWGS topics?

Textbook, Handouts and Information, Forums, Videos

- ▷ **No Textbook:** but lots of online python tutorials on the web.
- ▷ Course notes will be posted at <http://kwarc.info/teaching/IWGS> (see references)
 - ▷ I mostly prepare/adapt/correct them as we go along.
 - ▷ please e-mail me any errors/shortcomings you notice. (improve for the group)
- ▷ The lecture videos of WS 2020/21 are at <https://www.fau.tv/course/id/2350> (not much changed)
- ▷ Matrix chat at #iwgs:fau.de (via IDM) (instructions)
- ▷ **StudOn Forum:** <https://www.studon.fau.de/frm5075965.html> for
 - ▷ announcements, homeworks (my view on the forum)
 - ▷ questions, discussion among your fellow students (your forum too, use it!)
- ▷ If you become an active discussion group, the forum turns into a valuable resource!

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 210 2024-02-08

Next we come to a special project that is going on in parallel to teaching the course. I am using the course materials as a research object as well. This gives you an additional resource, but may affect the shape of the courses materials (which now serve double purpose). Of course I can use all the help on the research project I can get, so please give me feedback, report errors and shortcomings, and suggest improvements.

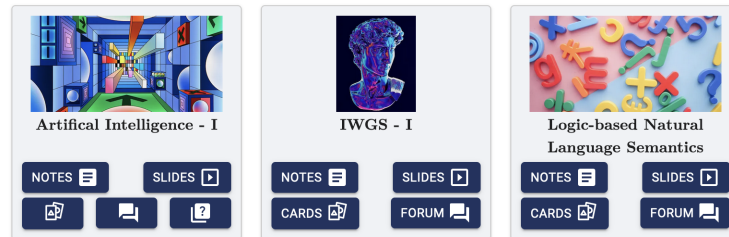
Experiment: Learning Support with KWARC Technologies

- ▷ **My research area:** Deep representation formats for (mathematical) knowledge
- ▷ **One Application:** Learning support systems (represent knowledge to transport it)
- ▷ **Experiment:** Start with this course (Drink my own medicine)
 1. Re-represent the slide materials in OMDoc (Open Mathematical Documents)
 2. Feed it into the ALeA system (<http://courses.voll-ki.fau.de>)

- 3. Try it on you all (to get feedback from you)
- ▷ Research tasks
 - ▷ help me complete the material on the slides (what is missing/would help?)
 - ▷ I need to remember “what I say”, examples on the board. (take notes)
- ▷ Benefits for you (so why should you help?)
 - ▷ you will be mentioned in the acknowledgements (for all that is worth)
 - ▷ you will help build better course materials (think of next-year’s students)

VoLL-KI Portal at <https://courses.voll-ki.fau.de>

- ▷ **Portal for ALeA Courses:** <https://courses.voll-ki.fau.de>

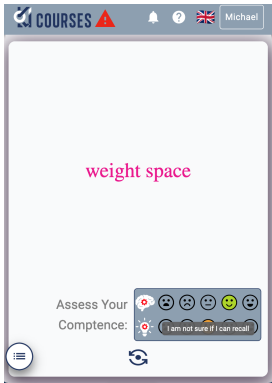
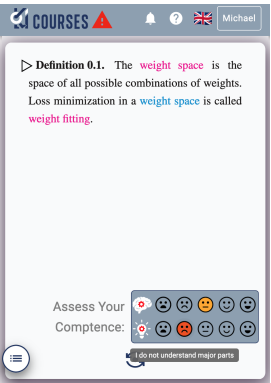


- ▷ **AI-1 in ALeA:** <https://courses.voll-ki.fau.de/course-home/ai-1>
 - ▷ All details for the course.
 - ▷ recorded syllabus (keep track of material covered in course)
 - ▷ syllabus of the last semester (for over/preview)
- ▷ **ALeA Status:** The **ALeA** system is deployed at FAU for over 1000 students taking six courses
 - ▷ (some) students use the system actively (our logs tell us)
 - ▷ reviews are mostly positive/enthusiastic (error reports pour in)

The VoLL-KI course portal (and the AI-1) home page is the central entry point for working with the ALeA system. You can get to all the components of the system, including two presentations of the course contents (notes- and slides-centric ones), the flash cards, the localized forum, and the quiz dashboard.



New Feature: Drilling with Flashcards

- ▷ **Flashcards** challenge you with a **task** (term/problem) on the **front**...

... and the definition/answer is on the *back*.

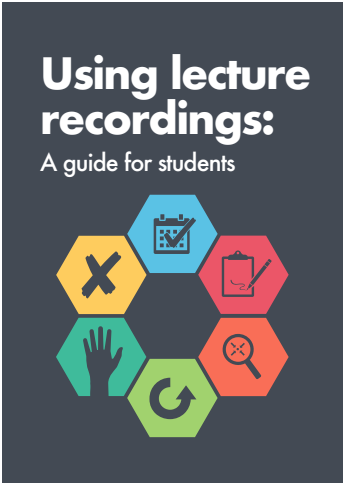
- ▷ Self-assessment updates the learner model (before/after)
- ▷ **Idea:** Challenge yourself to a *card stack*, keep drilling/assessing flashcards until the learner model eliminates all.
- ▷ **Bonus:** Flashcards can be generated from existing semantic markup (*educational equivalent to free beer*)








Michael Kohlhase: Inf. Werkzeuge @ G/SW 2
213
2024-02-08


We have already seen above how the learner model can drive the drilling with flashcards. It can also be used for the configuration of card stacks by configuring a domain e.g. a section in the course materials and a competency threshold.



Practical recommendations on Lecture Videos

- ▷ **Excellent Guide:** [Nor+18a] (german Version at [Nor+18b])



-  Attend lectures.
-  Take notes.
-  Be specific.
-  Catch up.
-  Ask for help.
-  Don't cut corners.

▷ Normally intended for "offline students" $\hat{=}$ everyone during Corona times.


Michael Kohlhase: Inf. Werkzeuge @ G/SW 2
214
2024-02-08


Software/Hardware tools

- ▷ You will need **computer** access for this course
- ▷ we recommend the use of standard software tools
 - ▷ find a **text editor** you are comfortable with (**get good with it**) A **text editor** is a program you can use to write **text files**. (not MSWord)
 - ▷ any **operating system** you like (I can only help with **UNIX**)
 - ▷ Any browser you like (I use **FireFox**: less spying)
- ▷ **Advice: learn how to touch-type NOW** (reap the benefits earlier, not later)
 - ▷ you will be typing multiple hours/week in the next decades
 - ▷ touch-typing is about twice as fast as "system eagle".
 - ▷ you can learn it in two weeks (good programs)

Touch-typing: You should not underestimate the amount of time you will spend typing during your studies. Even if you consider yourself fluent in two-finger typing, touch-typing will give you a factor two in speed. This ability will save you at least half an hour per day, once you master it. Which can make a crucial difference in your success.

Touch-typing is very easy to learn, if you practice about an hour a day for a week, you will re-gain your two-finger speed and from then on start saving time. There are various free typing tutors on the network. At http://typingsoft.com/all_typing_tutors.htm you can find about programs, most for windows, some for linux. I would probably try **Ktouch** or **TuxType**

Darko Pesikan (one of the previous TAs) recommends the **TypingMaster** program. You can download a demo version from <http://www.typingmaster.com/index.asp?go=tutordemo>

You can find more information by googling something like "learn to touch-type". (goto <http://www.google.com> and type these search terms).

Outline of IWGS-II:

- ▷ **Databases**
 - ▷ CRUD operations, **querying**, and python embedding
 - ▷ **XML** and **JSON** for file based data storage
- ▷ BooksApp: a Books Application with **persistent** storage
- ▷ **Image processing**
 - ▷ Basics
 - ▷ Image transformations, Image Understanding
- ▷ Ontologies, **semantic web**, and WissKI
 - ▷ Ontologies (inference \leadsto get out more than you put in)
 - ▷ **semantic web** Technologies (standardize ontology formats and inference)
 - ▷ Using **semantic web** Tech for cultural heritage research data \leadsto the WissKI System

- ▷ Legal Foundations of Information Systems
 - ▷ Copyright & Licensing
 - ▷ Data Protection (GDPR)

In IWGS-II, we want to consolidate the methods and technologies we learn in a small information system, which students build in groups, and which will serve as a running example for the course. These projects will consist of documents, data, and programs.

IWGS-II Project

- ▷ **Idea:** Consolidate the techniques from IWGS-I and IWGS-II into a prototypical information system for Art History @ FAU. (Practical Digital Humanities)
- ▷ **A Running Example:** Research image + metadata collection "Bauernkirmes" provided by Prof. Peter Bell



- ▷ **What will you do?:** Build a web-based image/data manager, test image algorithms, annotate ontologically, ...
- ▷ **How will we organize this:** Mostly via the group homework assignments (together they will make the project)

Chapter 9

Databases

We now come to one of the core tools of **computer science**: **databases** give us a means to store large collections of **data** and organize them for **efficient** access. We will introduce the underlying concepts by example, go over the basics of **relational database systems** and the **SQL** language, and experiment with a concrete system: **SQLite** and its embedding into **Python**. **Acknowledgements:** We have borrowed and adapted examples and from [SSU04] and [PMDA] in this chapter.

9.1 Introduction

Before we do anything else, we will look at various concepts around **data** to clarify concerns.

Databases, Data, Information, and Knowledge

- ▷ **Definition 9.1.1.** Discrete, objective facts or observations, which are unorganized and uninterpreted are called **data** (singular **datum**).
- ▷ According to Probst/Raub/Romhardt [PRR97]

The diagram illustrates the progression from Glyphs to Knowledge through Data and Information. It consists of four main stages in colored boxes: Glyphs (orange), Data (orange), Information (red), and Knowledge (purple), connected by arrows. Above each stage is a concept in a light blue circle, and below each is an example in a light blue circle. 1. Glyphs: Concept 'Character Set', Example '0', '9', '5', ' ', ' '. 2. Data: Concept 'Syntax', Example '0,95'. 3. Information: Concept 'Context', Example 'Exchange rate 1 \$ = 0,95 €'. 4. Knowledge: Concept 'Networking', Example 'Markt mechanisms concerning exchange rates'.

- ▷ **Example 9.1.2.** The height of Mt. Everest (8.848 meters) is a **datum**.
- ▷ **Definition 9.1.3.** A **database** is an organized collection of **data**, stored and accessed electronically from a **computer system**.
- ▷

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 218 2024-02-08

To get an intuition about the possibilities of storing **data**, we look at some common ways – some of which we have already seen – and characterize them by some practical dimensions.

Storing Data Electronically

- ▷ Four conventional ways of storing data: (mileage varies)
 - ▷ In the computer's memory (RAM) (very fast (+), random access (+), but not persistent (-))
 - ▷ In a text file (persistent (+), fast (+), sequential access (), unstructured ())

```

"ArtistId", "ArtistName"
1, "AC/DC"
2, "Louis Armstrong"
3, "Iron Maiden"
4, "Miles Davis"
5, "Pat Benetar"
6, "Stevie Ray Vaughan"
7, "Averged Sevenfold"
8, "Destiny's Child"
9, "Snoop Dogg"
  
```

- ▷ In a spreadsheet (persistent (+), 2D-structured (+-), relations (+), slow (-))

ArtistId	ArtistName
1	AC/DC
2	Louis Armstrong
3	Iron Maiden
4	Miles Davis
5	Pat Benetar
6	Stevie Ray Vaughan
7	Averged Sevenfold
8	Destiny's Child
9	Snoop Dogg

- ▷ In a database (persistent (+), scalable (+), relations(+), managed (+), slow (-))

ArtistId	ArtistName
1	AC/DC
2	Louis Armstrong
3	Iron Maiden
4	Miles Davis
5	Pat Benetar
6	Stevie Ray Vaughan
7	Averged Sevenfold
8	Destiny's Child
9	Snoop Dogg

- ▷ Databases constitute the most scalable, [persistent](#) solution.

We will study the practical aspects of one particularly important class of [database](#) systems: [relational database management systems](#).

9.2 Relational Databases

We will now study a particular kind of [database](#): [relational database](#), as these are the most widely used and structured ones.¹

EdN:1

(Relational) Database Management Systems

- ▷ **Definition 9.2.1.** A [database management system \(DBMS\)](#) is [program](#) that [interacts](#) with end [users](#), [applications](#), and a [database](#) to capture and analyze the [data](#) and provides facilities to administer the [database](#).
- ▷ There are different types of [DBMS](#), we will concentrate on [relational](#) ones.
- ▷ **Definition 9.2.2.** In a [relational database management system \(RDBMS\)](#), [data](#) are represented as [tables](#): every [datum](#) is represented by a [row](#) (also called [database record](#)), which has a [value](#) for all [columns](#) (also called an [column attribute](#)) or [field](#)). A [null value](#) is a special “[value](#)” used to denote a missing [value](#).
- ▷ **Remark:** [Mathematically](#), each [row](#) is an n [tuple](#) of values, and thus a [table](#) an n -ary [relation](#). (useful for standardizing RDBMS operations)
- ▷ **Example 9.2.3 (Bibliographic Data).**




LastN	FirstN	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY

- ▷ **Definition 9.2.4.** [Tables](#) are identified by [table name](#) and individual components of [records](#) by [column name](#).

As [RDBMS](#) constitute the backbone of of modern information technology, there are many many [implementations](#), commercial ones and open source ones as well. For our purposes, open-source systems are completely sufficient, so we list the most important ones here.

Open-Source Relational Database Management Systems

¹EdNOTE: MK: In the last years, [NoSQL databases](#) and [JSON](#) have gained prominaence. Intro them at the end and reference them here.

- ▷ **Definition 9.2.5.** MySQL is an open source RDBMS. For simple data sets and web applications MySQL is a fast and stable multi user system featuring an SQL database server that can be accessed by multiple clients. 
- ▷ **Definition 9.2.6.** PostgreSQL is an open source RDBMS with an emphasis on extensibility, standards compliance, and scalability. 
- ▷ **Definition 9.2.7.** SQLite is an embeddable RDBMS. Instead of a database server, SQLite uses a single database file, therefore no server configuration is necessary. 
- ▷ **Remark:** At the level we use SQL in IWGS, all are equivalent.
- ▷ We will use SQLite in IWGS, since it is easiest to install and configure.

Now that we have made our first steps in the SQL language and with RDBMS in general, let us pick a concrete RDBMS to experiment with.

Working with SQLite (via the SQLite shell)

- ▷ In IWGS we will use SQLite, since it is very lightweight, easy to install, but feature complete, and widely used.
- ▷ Download SQLite at <https://www.sqlite.org/download.html>,
 - ▷ e.g. sqlite-dll-win64-x64-3280000.zip for windows.
 - ▷ unzip it into a suitable location, start sqlite3.exe there
 - ▷ this opens a command line interpreter: the SQLite shell. (all DBs have one) test it with .help that tells you about more “dot commands”.

```
> sqlite3
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .help
.archive ... Manage SQL archives: ".archive --help" for details
.auth ON|OFF Show authorizer callbacks
[...]
```

- ▷ If you have a database file books.db from Example 9.3.8, use that.

```
> sqlite3 books.db
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
> .tables
Books
>select * from Books;
Twain|Mark|1835|1910|Huckleberry Finn|1986|Penguin USA|NY
Twain|Mark|1835|1910|Tom Sawyer|1987|Viking|NY
Cather|Willa|1873|1947|My Antonia|1995|Library of America|NY
Hemingway|Ernest|1899|1961|The Sun Also Rises|1995|Scribner|NY
```

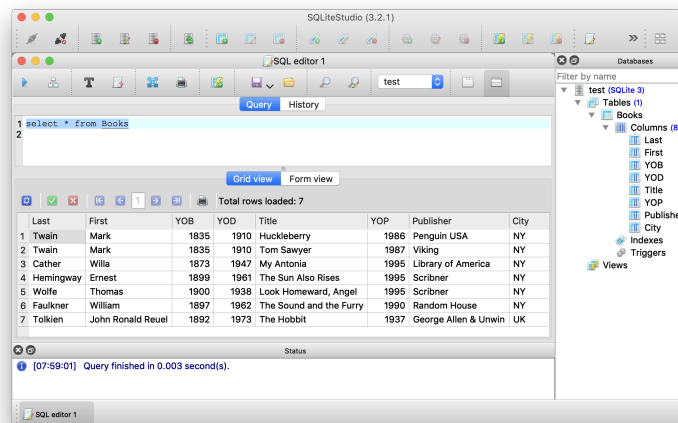
```
Wolfe|Thomas|1900|1938|Look Homeward, Angel|1995|Scribner|NY
Faulkner|William|1897|1962|The Sound and the Fury|1990|Random House|NY
Tolkien|John Ronald Reuel|1892|1973|The Hobbit|1937|George Allen Unwin|UK
```

- ▷ .tables shows the available **tables**
- ▷ `select * from Books` is **SQL** (see below); it shows all entries of the **Books table**.

Interacting with **SQLite** via the **database shell** is nice, but can be quite tedious. Fortunately, there are better alternatives.

A Graphical User Interface for SQLite

- ▷ **Definition 9.2.8.** A **database browser** is a **graphical user interface** for a **RDBMS** that (typically) bundles an **SQL instruction editor** with displays for **query results** and the **database schema** in separate **windows**.
- ▷ I will sometimes use one for **SQLite** in the slides: **SQLite Studio** (lots of others)
- ▷ download from <https://sqlitestudio.pl>



- ▷ Everything we can do with this, we can do with the **database shell** as well. (just looks nicer)

9.3 SQL – A Standardized Interface to RDBMS

Idea: To **interact** with **RDBMSs**, we need a language to describe **tables** to the system, so that they can be created, read, updated, and deleted. In fact while we are at it, we need a language for all **RDBMS** operations. The domain specific language **SQL** (pronounced like “sequel”) fills this need. It is internationally standardized, so that it can be used as the lingua franca for all **RDBMSs**, insulating users and application **programmers** against system internals.

SQL: The Structured Query Language

- ▷ **Idea:** We need a language for describing all operations of a **RDBMSs**.
 - ▷ **basics:** creating, reading, updating, deleting **database** components (**CRUD**)
 - ▷ **querying:** selecting from and inserting into the **database**
 - ▷ **access control:** who can do what in a **database**
 - ▷ **transactions:** ensuring a consistent **database** state.

Definition 9.3.1. **SQL**, the **structured query language** is a **domain-specific language** for managing **data** held in a **RDBMS**. **SQL instructions** are directly executed by the **RDBMS** to change the **database state** or compute **answers** to **SQL queries**.

▷

We start off with a fragment of **SQL** that is concerned with setting up the **database schema**, which gives structure to the **data** in the **database**. This **schema** is used by the **RDBMS** to optimize **database** access.

DDL: Data Definition Language

- ▷ **Definition 9.3.2.** The **data definition language (DDL)** is a subset of **SQL instructions** that address the creation and deletion of **database** objects.
- ▷ **Definition 9.3.3.** The **SQL** statement **CREATE TABLE**⟨⟨name⟩⟩ (⟨⟨coldefs⟩⟩) creates a **table** with name ⟨⟨name⟩⟩. ⟨⟨coldefs⟩⟩ are **column specifications** that specify the **columns**: it is a comma-separated list of **column names** and **SQL data type**. The totality of all **column specifications** of all **tables** in a **database** is called the **database schema**.
- ▷ **Example 9.3.4 (Creating a Table).** The following **SQL** statement creates the **table** from Example 9.2.3

```
CREATE TABLE Books (
  LastN varchar(128), FirstN varchar(128),
  YOB int, YOD int, Title varchar(255), YOP int,
  Publisher varchar(128), City varchar(128)
);
```

- ▷ Other **CREATE** statements exist, e.g. **CREATE DATABASE** ⟨⟨name⟩⟩.
- ▷ **Definition 9.3.5.** The **SQL** statement **DROP** ⟨⟨obj⟩⟩ ⟨⟨name⟩⟩ deletes the **database** object of class ⟨⟨obj⟩⟩ with name ⟨⟨name⟩⟩.

We have seen above that the **database schema** needs a **data type** for every **column**. We give an overview over the most important ones here.

SQL Data Types (for Column Specifications)

- ▷ **Definition 9.3.6.** **SQL** specifies **data type** for **values** including:
 - ▷ **VARCHAR** (⟨⟨length⟩⟩): character strings, including Unicode, of a variable length

- is up to the maximum length of `⟨length⟩`.
- ▷ **BOOL** truth values: **true**, **false** and case variants.
 - ▷ **INT**: Integers
 - ▷ **FLOAT**: floating point numbers
 - ▷ **DATE**: dates, e.g. **DATE** '1999–01–01' or **DATE** '2000–2–2'
 - ▷ **TIME**: time points in ISO format, e.g. **TIME** '00:00:00' or **time** '23:59:59.99'
 - ▷ **TIMESTAMP**: a combination of **DATE** and **TIME** (separated by a blank).
 - ▷ **CLOB** (`⟨length⟩`) (character large object) up to (typically) 2GiB
 - ▷ **BLOB** (`⟨length⟩`) (binary large object) up to (typically) 2GiB

We now come to the **SQL** commands for inserting content into the **database tables** we have created above. This is quite straight-forward.

SQL: Adding Records to Tables

- ▷ **Definition 9.3.7.** **SQL** provides the **INSERT INTO** command for **inserting** records into a **table**. This comes in two forms:

1. **INSERT INTO** `⟨table⟩` **VALUES** (`⟨vals⟩`); where `⟨vals⟩` is a comma-separated list of values given in the order the columns were declared in the **CREATE TABLE** instruction.
2. **INSERT INTO** `⟨table⟩` (`⟨cols⟩`) **VALUES** (`⟨vals⟩`) where `⟨vals⟩` is a comma-separated list of values given in the order of `⟨cols⟩` (a subset of columns) all other fields are filled with **NULL**

- ▷ **Example 9.3.8 (Inserting into the Books Table).** The given the **table** Books from Example 9.3.4 we can add a record with

```
INSERT INTO Books
VALUES ('Tolkien', 'John_Ronald_Reuel', 1892, 1973, 'The_Hobbit', 1937,
       'George_Allen_Unwin', 'UK');
```

- ▷ **Example 9.3.9 (Inserting Partial Data).** Using the second form of the **INSERT** instruction, we can insert partial **data**. (all we have)

```
INSERT INTO Books (FirstN, LastN, YOB, title, YOP)
VALUES ('Michael', 'Kohlhase', '1964', 'IWGS_Course_Notes', '2018');
```

With an insert facility, we need to be able to delete records as well, again it is straight-forward, with the exception that we have to identify which records to delete.

SQL: Deleting Records from Tables

- ▷ **Definition 9.3.10.** The **SQL delete** statement allows to change existing records.


```
DELETE FROM ⟨table⟩ WHERE ⟨condition⟩;
```

- ▷ **Example 9.3.11.** Deleting the record for “Huckleberry Finn”.

```
DELETE FROM Works WHERE Title = 'Huckleberry_Finn'
```

- ▷ ⚠ If we leave out the **WHERE** clause, all **rows** are deleted.
- ▷ **Note:** There is much more to the **WHERE** clause, we will get to that when we come to [SQL querying](#). (see [section 9.7](#))

And now we come to a variant of [database](#) insertion: record update. In principle, this could be achieved by deleting the record and then re-inserting the changed one, but the update instruction presented here is more [efficient](#).

SQL: Updating Records in Tables

- ▷ **Definition 9.3.12.** The **SQL update** statement allows to change existing records.

```
UPDATE <<table>>  
SET <<column>>_1 = <<value>>_1, <<column>>_2 = <<value>>_2, ...  
WHERE <<condition>>;
```

- ▷ **Example 9.3.13.** Updating the publisher in “Huckleberry Finn”.

```
UPDATE Books  
SET Publisher = 'Chatto/Windus', YOP = 1884, City = 'London'  
WHERE Title = 'Huckleberry_Finn'
```

- ▷ ⚠ If we leave out the **WHERE** clause, all **rows** are updated.

9.4 ER-Diagrams and Complex Database Schemata

We now come to a very important aspect of structured [databases](#): designing the [database schema](#) and with this determining the [data efficiency](#) and computational performance of the [database](#) itself. We get glimpse of the standard tool: [entity relationship diagrams](#) here.

Avoiding Redundancy in Databases

- ▷ Recall the [books table](#) from [Example 9.2.3](#):

LastN	FirstN	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY

- ▷ **Observation:** Some of the fields appear multiple times, e.g. “Mark Twain”.
- ▷ ⚠ When the [database](#) grows this can lead to scalability problems:

- ▷ in **querying**: e.g. if we look for all works by Mark Twain
- ▷ in **maintenance**: e.g. if we want to replace the pen name “Mark Twain” by the real name “Samuel Langhorne Clemens”.
- ▷ **Idea**: Separate concerns (here Authors, Works, and Publishers) into separate entities, mark their relations.
 - ▷ Develop a graphical notation for planning
 - ▷ **Implement** that into the **database**

After this discussion on why we need to design an **efficient database schema** to the **entity relationship diagram** themselves.

Entity Relationship Diagrams

- ▷ **Definition 9.4.1.** An **entity relationship diagram (ERD)** illustrates the logical structure of a **database**. It consists of **entities** that characterize (sets of) objects by their **attributes** and **relations** between them.
- ▷ **Example 9.4.2 (An ERD for Books).** Recall the Books table from Example 9.2.3:

LastN	FirstN	YOB	YOD	Title	YOP	Publisher	City
Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
Faulkner	William	1897	1962	The Sound and the Furry	1990	Random House	NY

- ▷ **Problem**: We have duplicate information in the authors and publishers
- ▷ **Idea**: Spread the Books information over multiple **tables**.

Authors

Last Name

First Name

Birth Date

Death Date

wrote

1 writ. by

Works

Title

PubDate

publ.

* publ. by

Publ

Name

City

Generally, a good **database** design is almost always worth the effort, since it makes the code and maintenance of the applications based on this **database** much simpler and intuitive.

We are fully aware, that this little example completely under-sells **entity relationship diagrams** and does not do this important topic justice. Fortunately, the DH students at FAU have the mandatory course “Konzeptuelle Modellierung” which does.

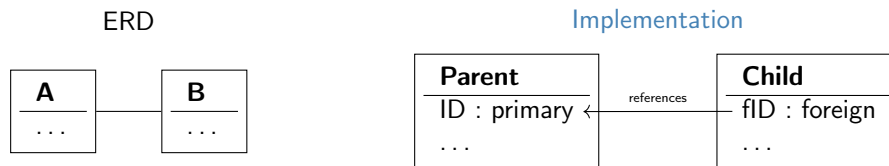
We now come to the **implementation** of the ideas from the **entity relationship diagrams**. The key idea is to have references between **tables**. These are mediated by special **database columns** types, which we now introduce.

Linking Tables via Primary and Foreign Keys

- ▷ **Definition 9.4.3.** A **column** in a **table** can be designated as a **primary key**, if its

values are **non-null** and **unique** i.e. all distinct.

- ▷ In **DDL**, we just add the keyword **PRIMARY KEY** to the **column specification**.
- ▷ **Definition 9.4.4.** A **foreign key** is a **column** (or collection of **columns**) in one **table** (called the **child table**) that refers to the **primary key** in another **table** (called the **reference table** or **parent table**).
- ▷ **Intuition:** Together **primary keys** and **foreign keys** can be used to link **tables** or (dually) to spread information over multiple **tables**.

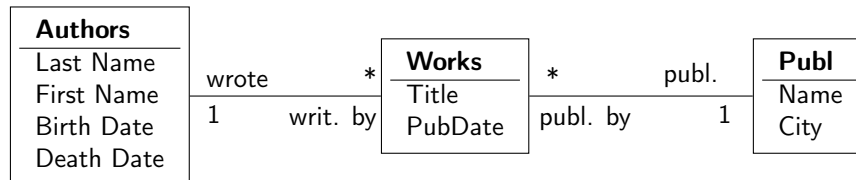


- ▷ **BTW:** **Primary keys** are great for identification in the **WHERE** clauses of **SQL** instructions.

We now fortify our intuition on **primary** and **foreign keys** by taking up Example 9.4.2 again.

Linking Tables via Primary and Foreign Keys (Example)

- ▷ **Example 9.4.5.** Continuing Example 9.4.2, we now **implement**



by introducing **primary keys** in the **Authors** and **Publishers** **tables** and referencing them by **foreign keys** in the **Works** **table**.

```
CREATE TABLE Authors (AuthorID int PRIMARY KEY,
  LastN varchar(128), FirstN varchar(128), YOB int, YOD int);
```

```
CREATE TABLE Publishers (PublisherID int PRIMARY KEY,
  Name varchar(128), City varchar(128));
```

```
CREATE TABLE Works (
  Title varchar(255), YOP int, AuthorID int, PublisherID int,
  FOREIGN KEY(AuthorID) REFERENCES Authors(AuthorID),
  FOREIGN KEY(PublisherID) REFERENCES Publishers(PublisherID));
```

Linking Tables via Primary and Foreign Keys (continued)

- ▷ **Example 9.4.6 (Inserting into the Works Table).** The given the `tables` Works Authors, and Publishers from Example 9.4.5 we can add a record with

```
INSERT INTO Authors VALUES (1, 'Twain', 'Mark', 1835, 1910);
INSERT INTO Publishers VALUES (1, 'Penguin USA', 'NY');
INSERT INTO Works VALUES ('Huckleberry Finn', 1986, 1, 1);

INSERT INTO Publishers VALUES (2, 'Viking', 'NY');
INSERT INTO Works VALUES ('Tom Sawyer', 1987, 1, 2);
```

Note: We have introduced new integer-typed `columns` for the `primary key` in the Authors and Publishers `tables`. In principle, we could have designated any existing `column` as a `primary key` instead, if we were sure that the entries are unique – in our case an unreasonable assumption, even for the publishers.

We have also chosen not to introduce a `primary key` in the Works `table`, which is probably a design mistake in the long run, because this would be very important to have for `deletions` and `updates`.

9.5 RDBMS in Python

Let us now see how we can `interact` with `SQLite` programmatically from `Python` instead of from the `SQLite` shell or the `database` browser.

Using SQLite from Python

- ▷ We will use the `PySQLite` package
 - ▷ `install` it locally with `pip install pysqlite` for `Python 3`.
 - ▷ use `import sqlite3` to import the library in your programs.
- ▷ Typical `Python` program with `sqlite3`:

```
import sqlite3
# Open database connection
db = sqlite3.connect(⟨⟨host⟩⟩,⟨⟨user⟩⟩,⟨⟨pass⟩⟩,⟨⟨DBname⟩⟩)
# prepare a cursor object using cursor() method
cursor = db.cursor()
# execute SQL commands using the execute() method.
cursor.execute("⟨⟨SQL⟩⟩")
⟨⟨dataprocessingcode⟩⟩
# make sure data reaches disk
db.commit()
# disconnect from server
db.close()
```

We will assume this as a wrapper for all code examples below.

The script schema shows the normal way of setting up the `interaction` with a `database` using `sqlite3`:

1. We first connect to the `database` by specifying the `database` file in which the `data` is kept. Normally, this will be file on the local file system, but we can also use a file that is available

on a remote host `⟨host⟩`. Of course, to write to this file will normally require [authentication](#), therefore `sqlite3.connect` also takes a user name `⟨user⟩` and a password `⟨pass⟩` as additional arguments. An alternative for the `⟨DBName⟩` argument is the string `:memory:` which results in an in-memory [database](#) (no [persistent](#) storage). The result of the `sqlite3.connect` function is a [database object](#) `db`.


2. Then we create a [cursor](#) object `cursor` (cf. slide 246 for more details) by using the `cursor` [method](#) of the [database object](#) `db`.
3. Then we execute [SQL instructions](#) via `cursor.execute` and do the [data](#) processing we need for our application.
4. To make sure that the changes we made to the [database](#) are actually reflected on disk in the [database](#) file `⟨DBName⟩`, we commit the changes to disk via `db.commit()`.
5. Finally, we close the [database](#) connection via the `db.close` [method](#) to make sure that all our changes have reached the [database](#) file.

We will now put this [schema](#) to use using Example 9.3.8 as a basis.

Creating Tables in Python

▷ **Example 9.5.1.** Creating the [table](#) of Example 9.3.4

```
import sqlite3
# our database file
database = "C:\\sqlite\\db\\books.db"
# a string with the SQL instruction to create a table
create = """CREATE TABLE Books (
    LastN varchar(128), FirstN varchar(128), YOB int, YOD int,
    Title varchar(255), YOP int, Publisher varchar(128), City varchar(128));"""
insert1 = """INSERT INTO Books
    VALUES ('Twain', 'Mark', '1835', '1910', 'Huckleberry Finn', '1986',
    'Penguin USA', 'NY');"""
insert2 = """INSERT INTO Books
    VALUES ('Twain', 'Mark', '1835', '1910', 'Tom Sawyer', '1987',
    'Viking', 'NY');"""
# connect to the SQLite DB and make a cursor
db = sqlite3.connect(database)
cursor = db.cursor()
# create Books table by executing the cursor
cursor.execute("DROP TABLE Books;")
cursor.execute(create)
cursor.execute(insert1)
cursor.execute(insert2)
db.commit() # commit to disk
db.close() # clean up by closing
```




FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

236

2024-02-08



SOME RIGHTS RESERVED

In this example we first create an [SQL instruction](#) as a string, so that we can give them as arguments to the `cursor.execute` method conveniently.

Note that `cursor.execute` only executes a single [SQL instructions](#) (for safety reasons; see slide 249 – why does this help there?).

Note that we drop the `Books` [table](#) before (re)creating it, to be sure that we have the right structure and avoiding errors, when we run the [Python](#) script above twice. An alternative would have been to use `CREATE TABLE IF NOT EXISTS`, which only creates the [table](#) if there is none. But in our example here, where we directly fill the [table](#), dropping any old [tables](#) with the name `Books` seems the right thing to do.

There is an issue that sometimes baffles beginners: I have created a [table](#), inserted lots of [data](#) into it, closed the [database](#), and the next time I connect to the [database](#), it is empty ~> very annoying.

To understand this phenomenon, we have to understand a bit more how [databases](#) like [SQLite](#) work and the tradeoffs face when working with such systems.

To commit or not to commit?

- ▷ **Recall:** [SQLite](#) computes with [tables](#) in [memory](#) but uses [files](#) for persistence.
- ▷ **Also Recall:** [Memory](#) access is 100-10.000 times as fast as [file](#) access.
- ▷ **Idea 1:** Keep [tables](#) in [memory](#), write to [file](#) only when necessary.
- ▷ **Idea 2:** Give the user/programmer control over when to write to [file](#)
 - ▷ `db = sqlite3.connect(⟨⟨file⟩⟩)` connects to `⟨⟨file⟩⟩`, but computes in [memory](#),
 - ▷ `db.commit()` writes in-memory changes to `⟨⟨file⟩⟩`.
- ▷ **Problem:** We can have multiple [database](#) connections to the same [database](#) file in parallel, there may be race conditions and conflicts.
- ▷ **Our Solution:** Commit often enough! (your responsibility/fault)
- ▷ **General Solution:** [RDBMS](#) offer [database transactions](#). (not covered in IWGS)
- ▷ **Lazy Solution:** Set the connection to [autocommit mode](#): (system decides)
`sqlite3.connect(⟨⟨file⟩⟩, isolation_level = None)`

Excursion: The general solution to the problem of accessing a [database](#) from multiple programs or processes in parallel is solved by a complex technology called [database transactions](#), which allow users' to define a sensible unit of work (via `begin/end` bracketing) called a [transaction](#) and makes sure that the process

- behaves as if the user's process has sole access to the [database](#) system for the duration of the [transaction](#) ([isolation](#))
- any changes made during the [transaction](#) can be rolled back if an error occurs during processing ([integrity](#)).

[Transactions](#) are an essential, but complex technology that is beyond the scope of the IWGS course. For our understanding, `db.commit` is essentially just the end bracket of a [transaction](#).

9.6 Excursion: Programming with Exceptions in Python

Before we go on, we discuss how we can deal with errors in [Python](#) flexibly, so that our [web application](#) will not drop into the [Python](#) level and present the user with a stack trace.

We first introduce what errors really are in the [Python](#) context and how they are [raised](#) and [handled](#). Then we look at what this means for our handling of [database](#) connections.

How to deal with Errors in Python

- ▷ **Theorem 9.6.1 (Kohlhase's Law).** *I can be an idiot, and I do make mistakes!*
- ▷ **Corollary 9.6.2.** *Programming languages need a good way to deal with all kinds of errors!*

▷ **Definition 9.6.3.** An **exception** is a special **Python object**. **Raising** an exception e terminates computation and passes e to the next higher level.

▷ **Example 9.6.4 (Division by Zero).** The **Python interpreter** reports unhandled exceptions.

```
>>> -3 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

▷ **Exceptions** are **first class citizens** in **Python**, in particular they

▷ are classified by their **classes** in a hierarchy.

▷ **exception classes** can be defined by the user (**they inherit from the Exception class**)

```
class DivByZero (Exception)
    pass
```

▷ can be **raised** when an abnormal condition appears

```
if denominator == 0 :
    raise DivByZero
else
    «computation»
```

▷ can be **handled** in a **try/except** block (there can be multiple)

```
try:
    «tentativecomputation»
except : «err»1, ..., «err»n :
    «errorhandling»
finally :
    «cleanup»
```

Let us now apply **Python exceptions** to our situation. Here the most important source of errors is the **database connection** step, where a **database** file might be missing or a remote host with the **database** file offline.

Playing it Safe with Databases

▷ **Observation 9.6.5.** *Things can go wrong when connecting to a **database!** (e.g. **missing file**)*

▷ **Idea:** **Raise exceptions** and **handle** them.

▷ **Example 9.6.6.** we encapsulate a **try/except** block into a function for convenience

```
import sqlite3
from sqlite3 import Error
def sql_connection():
    try:
        db = sqlite3.connect(':memory:')
        print("Connection is established: Database is created in memory")
    except Error :
```

```

    print(Error)
finally:
    db.close()

```

The `sqlite3` package provides its own [exceptions](#), which we import separately. Other errors can be [handled](#) in additional `except` clauses.

9.7 Querying and Views in SQL

So far we have created, filled, and possibly updated [databases](#), but we have not done anything useful with them. That is the realm of [querying](#) in [SQL](#), which we will now come to.

We will first cover [SQL querying](#) from a single [table](#). There are many variants of the `SELECT`/`FROM`/`WHERE` instruction. We explain the most commonly used ones.

SQL Querying: The SELECT Statement

- ▷ [SQL](#) uses the `SELECT` instruction for retrieving [data](#) from a [database](#).
- ▷ `SELECT` `⟨columns⟩` `FROM` `⟨table⟩` returns all records from `⟨table⟩` restricted to the [fields](#) from `⟨columns⟩`.
- ▷ **Definition 9.7.1.** We call a `SELECT` instruction a [query](#).
- ▷ **Example 9.7.2.** `SELECT Title, YOP FROM Books;`

```

Huckleberry Finn|1986
Tom Sawyer|1987
My Antonia|1995
The Sun Also Rises|1995
Look Homeward, Angel|1995
The Sound and the Fury|1990
The Hobbit|1937

```
- ▷ `SELECT DISTINCT` removes duplicate values
- ▷ `SELECT * FROM` `⟨table⟩` returns all records from `⟨table⟩`.
- ▷ `SELECT` `⟨columns⟩` `FROM` `⟨table⟩` `WHERE` `⟨cond⟩` returns all records that match condition `⟨cond⟩`
- ▷ **Example 9.7.3.** `SELECT FirstN, LastN FROM Books WHERE YOP = 1995;`

```

Willa|Cather
Ernest|Hemingway
Thomas|Wolfe

```
- ▷ `SELECT` `⟨columns⟩` `FROM` `⟨table⟩` `ORDER BY` `⟨columns⟩` orders the results by `⟨columns⟩`
- ▷ **Example 9.7.4.** Ordering can be ascending (`ASC`) or descending (`DESC`)
`SELECT FirstN, LastN FROM Books ORDER BY LastN ASC, YOP DESC;`

There are some more variants, for instance we can add a **GROUP BY** clause, which allows to group the result *table* according to various conditions. We now generalize *SQL queries* by combining multiple *tables* into a virtual *table* from which we aggregate the results. Joins over that combine multiple *tables* in *queries* are the technique that allows to split *data* into multiple *tables* in the first place: we can re recreate the “original big *table*” via a *query*. We will restrict ourselves to the simplest kind of *table* join: the “inner join” below. There are quite a few variants of joins; we refer the reader to the literature on them.

Joining Tables in Queries

▷ **Problem:** We can *query* single *tables*, how cross-table *queries*? E.g. in

Authors
AuthorID ←
Last Name
First Name
Birth Date
Death Date

Works
Title
PubDate
AuthorID
PublisherID

Publishers
→ PublisherID
Name
City

▷ **Idea:** Virtually join *tables* for the *query*! (as if we had the large *books* table)

▷ **Definition 9.7.5.** A *table join* (or simply *join*) is a means for combining *columns* from one (*self join*) or more *tables* by using *values* common to each.

▷ **Example 9.7.6.** *Joining* all three *tables* from Example 9.4.2.

```

SELECT
  Authors.LastN, Authors.FirstN, Authors.YOB, Authors.YOD,
  Title, YOP, Publishers.Name, Publishers.City
FROM
  Works
INNER JOIN Authors ON Authors.AuthorID = Works.AuthorID
INNER JOIN Publishers ON Publishers.PublisherID = Works.PublisherID

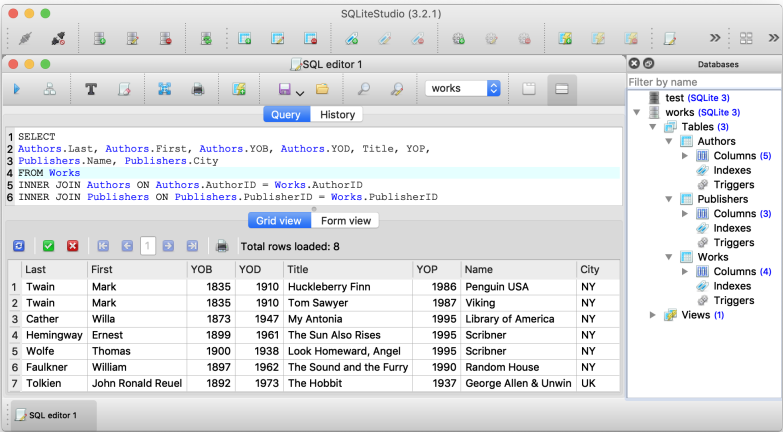
```

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 241 2024-02-08

The key idea in the *query* in Example 9.7.6 are the **join** statements in the last two lines. They do two things: first they tell *SQL* to extend the *Works* *table* with *data* from the two *tables* *Authors* and *Publishers*, and second they tell *SQL* how the extension should work: by making sure that in the extension the records in the *Works* *table* are extended with the (unique!) record in the *Authors* *table*, that has the same *AuthorID*, and analogously for the records from the *Publishers* *table*. Thus the two joins *implement* the two arrows in the *ER diagram* at the top of the slide. The result of this *query* is displayed on the next slide.

Joining Tables in Queries (Result)

▷ **Example 9.7.7.**



The screenshot shows the SQLiteStudio interface with a SQL query in the editor and its result grid. The query is:

```

1 SELECT
2 Authors.Last, Authors.First, Authors.YOB, Authors.YOD, Title, YOP,
3 Publishers.Name, Publishers.City
4 FROM Works
5 INNER JOIN Authors ON Authors.AuthorID = Works.AuthorID
6 INNER JOIN Publishers ON Publishers.PublisherID = Works.PublisherID

```

The result grid shows 8 rows of data:

Last	First	YOB	YOD	Title	YOP	Name	City	
1	Twain	Mark	1835	1910	Huckleberry Finn	1986	Penguin USA	NY
2	Twain	Mark	1835	1910	Tom Sawyer	1987	Viking	NY
3	Cather	Willa	1873	1947	My Antonia	1995	Library of America	NY
4	Hemingway	Ernest	1899	1961	The Sun Also Rises	1995	Scribner	NY
5	Wolfe	Thomas	1900	1938	Look Homeward, Angel	1995	Scribner	NY
6	Faulkner	William	1897	1962	The Sound and the Fury	1990	Random House	NY
7	Tolkien	John Ronald Reuel	1892	1973	The Hobbit	1937	George Allen & Unwin	UK

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 242 2024-02-08

Note that the result of the [query](#) from [Example 9.7.6](#) shown in [Example 9.7.7](#) exactly recreates the original big Books [table](#) from [Example 9.2.3](#). So we see that we have “lost nothing” by separating the [data](#) into three more [efficient](#) and less redundant – [tables](#).

We have seen above that we can [join](#) physical [database tables](#) to larger virtual ones whenever we need it in a [SQL query](#). This is good, but it can be made even better. [RDBMS](#) allow to persist virtual [table](#) in the [database schema](#) itself as [views](#).

Database Views: Persisting Queries

- ▷ **Observation:** Via the [join](#) in [Example 9.7.6](#), the Works [table queries](#) like the original Books [table](#).
- ▷ **Wouldn't it be nice** If we could also insert/update into that?
- ▷ **Definition 9.7.8.** A [database view](#) (or simply [view](#)) is a virtual [table](#) based on the result set of a [query](#). A [view](#) contains [rows](#) and [columns](#), just like a real [table](#). The [field](#) in a [view](#) are [fields](#) from one or more real [tables](#) in the [database](#).
- ▷ **Remark 9.7.9.** In many [RDBMS](#) we can even [insert](#), [delete](#), and [update](#) records in a [view](#), just as in any other [table](#) of the [database](#).
The [RDBMS](#) achieves this by automatically translating any change to the [view](#) into a set of changes to the underlying physical [tables](#).
- ▷ ⚠ but not in [SQLite](#). (this is an omission due to simplicity)

Remark: With [views](#) we can “have our cake and eat it too”: We can make our [database schema](#) space [efficient](#) by removing redundancies using “small [tables](#)” and still have our “big [tables](#)” that make our life convenient e.g. when [inserting](#) records. Consider our Books example again: we can give the [query](#) from [Example 9.7.6](#) a name and let the [RDBMS](#) treat it as a (virtual) [table](#).

Database Views: Persisting Queries (Books Example)

- ▷ **Example 9.7.10.** Use the [query](#) from Example 9.7.6 to define a view

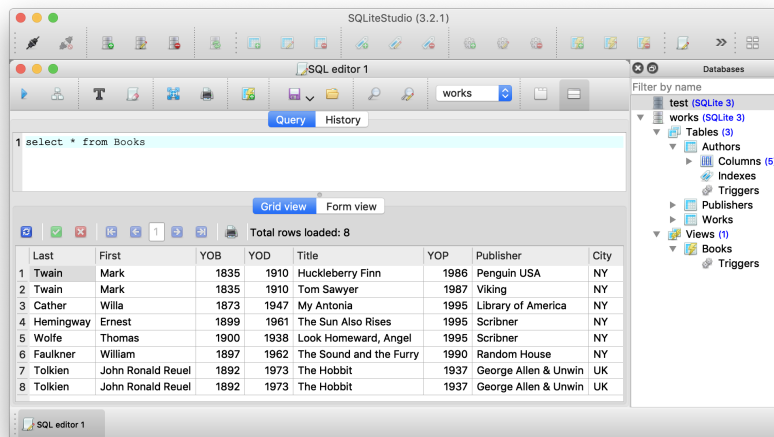
```
CREATE VIEW Books AS
SELECT
  Authors.LastN AS LastN, Authors.FirstN AS FirstN,
  Authors.YOB AS YOB, Authors.YOD AS YOD,
  Title, YOP,
  Publishers.Name AS Publisher, Publishers.City AS City
FROM
  Works
INNER JOIN Authors ON Authors.AuthorID = Works.AuthorID
INNER JOIN Publishers ON Publishers.PublisherID = Works.PublisherID
```

Use AS clauses in SELECT to specify [column names](#).

The proof is in the pudding. We see that `Books` [view](#) behaves exactly like the big (unstructured) books table from above. On the right of the [database browser window](#) we can see that it is actually a [view](#).

Database Views: Persisting Queries (Books Example)

- ▷ **Example 9.7.11.**



9.8 Querying via Python

Now it is time to turn to understanding [querying programmatically](#) in [Python](#). The main concept to grasp is that of a [cursor](#).

Working with Cursors

- ▷ **Definition 9.8.1.** A [cursor](#) is a named object that encapsulates a set of [query results](#) in a (virtual) [database table](#).

- ▷ To work with a `cursor` in `sqlite3`,
 - ▷ create a `cursor object` via the `cursor` method of your `database object`.
 - ▷ Open the cursor to establish the result set via its `execute` method
 - ▷ Fetch the `data` into local variables as needed from the `cursor`.
- ▷ The `cursor` class in `sqlite3` provides additional methods:
 - ▷ `fetchone()`: return one row as an array/list
 - ▷ `fetchall()`: return all rows a list of lists.
 - ▷ `fetchsome(⟨n⟩)`: return `⟨n⟩` rows a list of lists.
 - ▷ `rowcount()`: the number of `rows` in the `cursor`
- ▷ **Intuition:** `Cursors` allow `programmers` to repeatedly use a `database query`.

Again, we fortify our intuitions by making a little example: we pretty-print the some of the information by looping over result of fetching all the records from a given cursor.²

Extended Example: Listing Authors from the Books Table

▷ Example 9.8.2.

```
sql = 'SELECT FirstN, LastN, YOB FROM Books WHERE YOD < 1950;'
cursor.execute(sql)
print('There are', cursor.rowcount, 'books whose authors died before 1950:\n')
for row in cursor.fetchall():
    print(row[0], row[1], row[2], row[3], '\n')
print('That is all; if you want more, add more to the database!')
```

If we have a large number of uniform `SQL instructions`, then we can bundle them, by iterating over a list of `parameters`. In the example below, we explicitly write down the list, but in applications, the list would be e.g. read from a `metadata file`.

Inserting Multiple Records (Example)

- ▷ The `cursor.executemany` method takes an `SQL instruction` with `parameters` and a list of suitable tuples and executes them.
- ▷ **Example 9.8.3.** So the final form of insertion in Example 9.5.1 would be to define variable with a list of book tuples:

```
booklist = [
    ('Twain', 'Mark', 1835, 1910, 'Huckleberry_Finn', 1986, 'Penguin_USA', 'NY'),
    ('Twain', 'Mark', 1835, 1910, 'Tom_Sawyer', 1987, 'Viking', 'NY'),
    ('Cather', 'Willa', 1873, 1947, 'My_Antonia', 1995, 'Library_of_America', 'NY'),
    ('Hemingway', 'Ernest', 1899, 1961, 'The_Sun_Also_Rises', 1995, 'Scribner', 'NY'),
    ('Wolfe', 'Thomas', 1900, 1938, 'Look_Homeward_Angel', 1995, 'Scribner', 'NY'),
    ('Faulkner', 'William', 1897, 1962, 'The_Sound_and_the_Fury', 1990, 'Random_House', 'NY'),
    ('Tolkien', 'John_Ronald_Reuel', 1892, 1973, 'The_Hobbit', 1937, 'George_Allen_Unwin', 'UK')]
```

²EDNOTE: MK: show the results

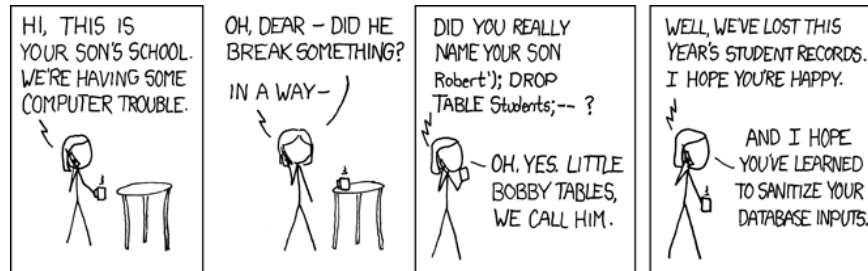
and then insert it via a call of `cursor.executemany`:

```
cursor.executemany('INSERT INTO Books VALUES(?,?,?,?,?,?,?)',booklist)
```

Now that we understand how to deal with `databases` programmatically, we can come to a real-world menace: `SQL injection attacks`. A large portion of the “hacking” events, where a `database` is taken over by malicious agents are based – at least in part – on such a technique. Therefore it is important to understand the basic principles involved, if only to understand how we can safeguard against them – see e.g. slide 250 below.

Beware of the Python/SQLite Interaction

- ▷ **What have we learned?:** At least you now understand the following web comic: (<https://xkcd.com/327/>)



- ▷ **Definition 9.8.4.** We call this an `SQL injection attack`.
- ▷ **Hint:** Imagine a `web application` where you add student names for enrolment.

```
name = input("Please enter student name: ")
cursor.execute(f"INSERT INTO Students VALUES (...,{Name},...);")
```

For the input `Robert'); DROP TABLE Students;` this has a `Python line` generates and executes the `SQL` instructions

```
INSERT INTO Students VALUES (... , 'Robert'); DROP TABLE Students;
```

Now we can understand why the restriction of `cursor.execute` to only one `SQL instruction` enhances security of the code: The hypothetical `cursor.execute('INSERT...')` command expects one `instruction`, but with the parameter substitution in the `f` string gets two. This would have raised an error and saved the school administration.

Finally we come back to the topic of preventing `SQL injection attacks`. We had seen that these occur when we build the argument string for a `cursor.execute` call. While the single-instruction-restriction of is some help, it is not enough. We essentially have to remove all the `SQL instructions` from any input string we substitute with. Fortunately, `SQL` is standardized, so we can `implement` that once and for all.

SQLite3 Parameter Substitution

- ▷ **Observation 9.8.5.** *We often need variables as parameters in `cursor.execute`.*

▷ **Example 9.8.6.** In Example 9.8.2 we can ask the user for a year.

▷ **The python way** would be to use `f strings`

```
year = input('Books, whose author died before what year?')
sql = f'SELECT FirstN, LastN, YOB FROM Books WHERE YOD < {year}'
cursor.execute(sql) # ⚠ never use f-strings here --> insecure
```

But this leads to vulnerability by `SQL injection attacks`. (↪ `Bobby Tables`)

▷ **Definition 9.8.7.** `sqlite3` supplies a `parameter substitution` that `SQL sanitizes` parameters (removes problematic `SQL instructions`).

▷ **The sqlite3 way** uses `parameter substitution` (multiple ? possible ↪ `tuple`)

```
year = input('Books, whose author died before')
select = 'SELECT Title FROM Books WHERE YOD < ?'
cursor.execute(select, (year,))
```

or in the “named style” ↪ `order-independent` (argument is a `dictionary`)

```
century = input('Century of the books?')
select = 'SELECT Title, YOP FROM Books WHERE YOP <= :start AND YOP > :end'
datadict = {'start': (century - 1) * 100, 'end': century * 100}
cursor.execute(select, datadict)
```

9.9 Real-Life Input/Output: XML and JSON

We have seen how we can use `Python` programs to fill `database` tables programmatically. But the treatment above did not map the dominant data management practices. In practice, `databases` are filled from various information sources, usually `CSV`, `XML`, and `JSON` files. Conversely, the `data` from a `database` is often exported to the same file formats for backup and/or communication.

To show the practices, we will see how to import `data` from an `XML` file into a data base, and how to export `data` as `JSON` in `Python`; the latter is an important technique for modern `web applications`.

Filling a DB from via XML (Specification)

▷ **Idea:** We want to make a `database` based `web application` for NYC museums.

▷ **Recall** the public catalog from Example 4.5.4 (Introduction to XML) in the IWGS lecture notes, the `XML` file is online at

<https://data.cityofnewyork.us/download/kcrm-j9hh/application/xml>

```
<?xml version="1.0" encoding="UTF-8"?>
< museums >
  < museum >
    < name >American Folk Art Museum</ name >
    < phone >212-265-1040</ phone >
    < address >45 W. 53rd St. (at Fifth Ave.)</ address >
    < closing >Closed: Monday</ closing >
    < rates >admission: $9; seniors/students, $7; under 12, free</ rates >
    < specials >
      Pay-what-you-wish: Friday after 5:30pm;
      refreshments and music available
    </ specials >
```

```

</museum>
<museum>
  <name>American Museum of Natural History</name>
  <phone>212-769-5200</phone>
  <address>Central Park West (at W. 79th St.)</address>
  <closing>Closed: Thanksgiving Day and Christmas Day</closing>

```

- ▷ **Idea:** We need [Python](#) program that
 - ▷ provides a [SQLite database](#) with a [table](#) 'museum' with columns 'name', 'phone', ..., 'specials' of appropriate type
 - ▷ reads the [XML](#) file from the [URL](#) above and fills the [table](#).
- ▷ **Possible Enhancement:** Encapsulate the functionality into a function, then we could run this program each night and keep the [database](#) up to date.

Let us actually [implement](#) this idea – after all, we have already seen all the building blocks already. The program itself is relatively straightforward; we go through the process step by step.

Filling a DB from via XML (Implementation)

- ▷ **Libraries:** `urllib` [UL] to retrieve the file and `lxml` [LXML] to [parse](#) it.

```

from lxml import etree
from urllib.request import urlopen
url = 'https://data.cityofnewyork.us/download/kcrm--j9hh/application/xml'
document = urlopen(url).read()
tree = etree.fromstring(document)

```

We now have a (large) [XML](#) tree in `tree`!

- ▷ Collect all the XML tags in all the museums (for the column names)

```

tags = []
for museum in tree:
    for info in museum:
        if info.tag not in tags:
            tags.append(info.tag)

```

- ▷ We create the [SQLite database](#) as discussed in slide 236.

- ▷ Then we assemble a [table](#) specification in a string `columns`:

```

columns = ""
for cn in tags:
    # All columns have their name and type TEXT
    columns += f",{cn}_TEXT"

```

- ▷ Create the Museums [table](#) from the specification in `columns`

```

cursor.execute("DROP TABLE IF EXISTS Museums;")
cursor.execute(f"""CREATE TABLE Museums
                (Id INTEGER PRIMARY KEY {columns});""")

```

- ▷ Now the most important part: We fill the [database](#)

```

for museum in tree:
    # Find and sanitise the contents of all child nodes of this museum.
    values = []
    for tag in tags:

```

```

        if museum.find(tag) != None:
            values.append(str(museum.find(tag).text).strip())
        else:
            values.append("-")

    # Insert the data for this museum into the database.
    cols = str(tuple(tags))

    # We need a tuple of one ? for each column.
    vals = "(" + ("?, " * len(tags))[:-2] + ")"

    insert = f"INSERT INTO Museums {cols} VALUES {vals}"
    cursor.execute(insert, tuple(values))

```

▷ We finalize the transaction as discussed in slide 236.

If we want to get a sense of the size and complexity of the complete program, we can look at it in one pice here:

The complete code in one block – a mere 51 lines

```

import sqlite3
from lxml import etree
from urllib.request import urlopen

# Download the XML file and Parse it
url = 'https://data.cityofnewyork.us/download/kcrm-j9hh/application/xml'
document = urlopen(url).read()
tree = etree.fromstring(document)

# First run-through of the XML: Collect the info types there,
tags = []
for museum in tree:
    for info in museum:
        if info.tag not in tags:
            tags.append(info.tag)

# Next, create database accordingly. First assemble a columns string.
columns = ""
for cn in tags:
    # All columns have their name and type TEXT
    columns += f",{cn} TEXT"

# Then, make the Museums table using that.
db = sqlite3.connect("./museums.sqlite")
cursor = db.cursor()
cursor.execute("DROP TABLE IF EXISTS Museums;")
cursor.execute(f"""CREATE TABLE Museums
                (Id INTEGER PRIMARY KEY {columns});""")

# Lastly, fill database.
for museum in tree:
    # Find and sanitise the contents of all child nodes of this museum.
    values = []
    for tag in tags:
        if museum.find(tag) != None:
            values.append(str(museum.find(tag).text).strip())
        else:
            values.append("-")

    # Insert the data for this museum into the database.

```

```

cols = str(tuple(tags))

# We need a tuple of one ? for each column.
vals = "(" + ("?, " * len(tags))[:-2] + ")"


insert = f"INSERT INTO Museums_{cols} VALUES_{vals}"
cursor.execute(insert, tuple(values))

# Finalise Transaction
db.commit()
db.close()

```


We will use the output direction of the envisioned museums [web application](#) to introduce another standard data representation format: [JSON](#) the preferred data interchange format for [web applications](#).

JSON — JavaScript Object Notation

- ▷ **Definition 9.9.1.** [JSON](#) ([JavaScript Object Notation](#)) is an open standard file format for interchange of structured [data](#). [JSON](#) uses human readable text to store and transmit [data](#) objects consisting of attribute–value pairs and sequences.
- ▷  [JSON](#) is very flexible, there need not be a regularizing schema.
- ▷ **Intuition:** [JSON](#) is for [JavaScript](#) as (nested) [dictionaries](#) are for [Python](#).
 - ▷ The browser can directly read [JSON](#) and use it via [JavaScript](#).
 - ▷ \leadsto [AJAX](#) $\hat{=}$ [JavaScript](#) can [query](#) the backend for [JSON data](#) to update parts of the [DOM](#). ([lightweight interaction](#))
- ▷ **Consequence:** [JSON](#) is the dominant interchange format for [web applications](#).
- ▷ **Another Intuition:** [JSON](#) objects are like [database records](#), but less rigid.
- ▷ **Idea:** Build a special [JSON database](#). ([JSON I/O](#); [efficient storage](#))
- ▷ **Definition 9.9.2.** [mongoDB](#) is the most popular [NoSQL database](#) system. ([no SQL inside](#))

As always, we will now look at how we can deal with with the newly introduced concept in [Python](#). As always there is a special library that does nearly all the work; here it is (obviously named) [json](#) library. It smoothes over the syntactic differences between [Python](#) dictionaries and [JSON](#) objects.

Dealing with JSON in Python

- ▷  Even though [JSON](#) concepts and syntax are similar to [Python](#) dictionaries, there are (subtle) differences.
- ▷ **Concretely:** [Python](#) allows more data types in dictionaries, e.g.

Python	JSON equivalent
True	true
False	false
float	Number
int	Number
None	null
dict	Object
list	Array
tuple	Array

▷ But these differences are systematic and can be overcome via the json library [JS].

▷ `json.dumps(⟨dict⟩)` takes a Python dictionary dict, produces a JSON string.

▷ `json.loads(⟨json⟩)` takes a JSON string json, produces a Python dictionary.

There are many ways to control the output (pretty-printing), see [JS].

We now give an JSON export program for the NYC Museums database for reference. All the technologies in this program have been covered above, so we just show it for self-study.

JSON Output for the NYC Museums DB

▷ **Libraries:** json for JSON [JS] and sqlite3 for the database.

```
import json
import sqlite3
```

▷ Connect to the SQLite database as usual and query the database for everything

```
db = sqlite3.connect("./museums.sqlite")
cursor = db.cursor()
cursor.execute("SELECT * FROM Museums;")
```

▷ Initialize a dictionary and the list of Museums column names

```
data = {}
data['museums'] = []
columns = ['name', 'phone', 'address', 'closing', 'rates', 'specials']
```

▷ For each of the rows in the Museums table build a row dictionary

```
for row in cursor.fetchall():
    # Generate a dictionary with columns as keys and entries as values.
    rowdict = { columns[n] : row[n] for n in range(6) }
    # Add that dictionary to the JSON data structure.
    data['museums'].append(rowdict)
```

▷ Dump the data dictionary as JSON into a file

```
with open('museums.json', 'w') as outfile:
    json.dump(data, outfile)
```

▷ Close the database as usual.

We set the list variable columns manually for convenience. Of course, if the database schema

should change, we have to adapt columns in our programs. Therefore a better way to handle this would be to get this information from the `database` itself, which we could do by the following command in SQLite:

```
PRAGMA table_info(Museums);
```

In our case, this gives us the following string, from which we can retrieve the information we need by a simple regular expression.

```
0|id|INTEGER|0|1
1|name|TEXT|0|0
2|phone|TEXT|0|0
3|address|TEXT|0|0
4|closing|TEXT|0|0
5|rates|TEXT|0|0
6|specials|TEXT|0|0
```

But note that the `PRAGMA` instruction is specific to SQLite; other systems have other ways of getting to this information.

JSON Output for the NYC Museums DB

```
import json
import sqlite3

# Connect to database and query database for everything.
db = sqlite3.connect("./museums.sqlite")
cursor = db.cursor()
cursor.execute("SELECT * FROM Museums;")

# Setup soon-to-be JSON dictionary and the necessary columns
data = {}
data['museums'] = []
columns = ['name', 'phone', 'address', 'closing', 'rates', 'specials']

# For every row in the result, do the following:
for row in cursor.fetchall():
    # Generate a dictionary with columns as keys and entries as values.
    rowdict = { columns[n] : row[n] for n in range(6) }

    # Add that dictionary to the JSON data structure.
    data['museums'].append(rowdict)

# Write collected JSON data to file.
with open('museums.json', 'w') as outfile:
    json.dump(data, outfile)

# Close database
db.close()
```

And now we can see the result of this export – at least an initial fragment for space reasons.

JSON Example (NYC Museums)

- ▷ **Example 9.9.3.** The NYC museums `data` from Example 4.5.4 (Introduction to XML) in the IWGS lecture notes as `JSON`:

We represent the `data` as a “sequence” of (nested) “dictionaries”

```
[
  {
    "name": "American Folk Art Museum",
    "phone": "212-265-1040",
    "address": "45 W. 53rd St. (at Fifth Ave.)",
```



```

    "closing": "Closed: Monday",
    "rates": {
      "admission": "$9",
      "seniors/students": "$7",
      "under 12": "free",
    }
    "specials": "Pay-what-you-wish: Friday after 5:30pm;
      refreshments and music available"
  }
  "name": "American Museum of Natural History",
  "phone": "212-769-5200",
  "address": "Central Park West (at W. 79th St.)"
  "closing": "Closed: Thanksgiving Day and Christmas Day"
  "rates": {
    "admission": "$16",
    "seniors/students": "$12",
    "kids 2-12": "$9",
    "under 2": "free"
  }
}
...
]

```

9.10 Exercises

Problem 10.1 (Setting up the Database)

In this exercise we will set up our [database](#) tables. Start by cloning the KirmesDH repository¹. The dataset consists of a directory `img/`, which contains [images](#) and a [folder metadata/](#) containing CSV files. The other directories are not important for this assignment.

Familiarize yourself with the [metadata](#) format. As you can see most files employ the same columns, however some [data](#) may be missing. We will mirror the given column structure in our [database](#).

1. In the given code skeleton, change the values of the variables `metadataFolder` and `imageFolder` at the top of the file according to your folder structure.
2. Establish a connection to the [database](#). Use the `databaseName` variable.
3. Create a table with name `Images` in the [database](#) with the following column structure:
 - `FileName`, type TEXT
 - `Title`, type TEXT
 - `Subtitle`, type TEXT
 - `Archive`, type TEXT
 - `Artist`, type TEXT
 - `Location`, type TEXT
 - `Date`, type TEXT
 - `Genre`, type TEXT
 - `Material`, type TEXT
 - `Url`, type TEXT
 - `Content`, type BLOB

¹<https://gitlab.cs.fau.de/iwgs-ss19/KirmesDH>

- At the end of the file, commit all changes you made to the `database` and close it.

Run your script and open the resulting `database` file in the DB Browser for SQLite. Make sure that you see the `Images` table and that its layout is correct.

Problem 10.2 (Parsing the Input Data)

In this exercise we will `parse` the `metadata files` and extract all relevant `data`. Since the input `data` is not curated very carefully and some entries may be missing, we need to design our program as robustly as possible.

Amend the `parseMetadata` function in the given `Python` script for this assignment. The prepared code opens the CSV file and uses the `csv` library to `parse` it. Detailed information on the `csv.DictReader` can be found here: <https://docs.python.org/3/library/csv.html#csv.DictReader>.

In the loop do the following for each row of the file:

- Use the `getValue` function to extract the relevant `data`.
- Call the `addImage` function with the `data`.

Make sure that the `data` is `parsed` correctly by running your `program` and printing the extracted values. Assure that the program does not crash if certain `data` fields are not available.

Problem 10.3 (Inserting Data into the Database)

In this last exercise we fill our `database` with the `parsed data`. Before starting with this task, assure that the previous two assignments work correctly.

Complete the `addImage` function.

- Check whether in the `img/` folder a file with the specified file name exists. If yes, open and read it and store the content in the `imageData` variable.
- Insert all data fields into the `database` by issuing the correct SQL command.

Run your script. Make sure it does not crash and check your `database` in the *DB Browser*. All values should be in the correct `column`. Some `rows` should have values in the `Content` column. In the *DB Browser* you can see the `image` when you click on the table cell.

We will now start establishing a `web server`, using the `bottle` framework we introduced last semester. We are building on top of the code above, so you may either continue with your own code or use the sample solution from last week as a starting point for this exercise.

For the `web server` we again prepared a code skeleton for you (`Server_Skeleton.py` and `Index_Skeleton.tpl`).

Problem 10.4 (Adding a Primary Key to our Table)

Our table `Images` from last week supports nearly all functionality we need. However currently it lacks the ability to uniquely identify a single entry, since all properties could be featured in multiple entries.

We therefore introduce `primary keys`. To this end, amend your `Images` table by adding a field `Id` of type `INTEGER`. Mark it as a `primary key`. When inserting into your `database`, you don't actually have to provide a value for the `Id`, since SQLite will simply use the next free number.

Problem 10.5 (Setting up our Web Server)

We will now set up a simple `web server` using the `bottle` framework. As a starting point you can use the `Server_Skeleton.py` and `Index_Skeleton.tpl` we provide you.

You might need to `install` the `bottle` package first. In your command prompt (terminal) issue the following command:

```
pip install bottle
```

You should now be able to run the provided code. Make sure you adapt the value of the variable `database` to match your `database` file.

After starting you can access your website by visiting the URL `http://localhost:8080/` in your browser. The content of this page is for you to **implement**.

We provide a `route /imageraw` in the `getImage` function. Follow the instructions in the code to try out the function and see how it works. For all operations which need to display `images` from the `database` on your website you should use this route.

Your job is to **implement** the `index` function, which is called when the home page is visited. In the end this page should display a large table where all entries of your `database` are listed.

1. Start by **querying** your `database` for the `data` you want to display. Select at least the `Id`, `Title`, `Subtitle`, `Artist`, `Material` and `Archive` of each entry. Issuing the appropriate SQL command should provide you a large list of entries. Make sure that this works before continuing.
2. Last semester we created websites in `bottle` by creating `HTML` code from python. This does not scale well to larger projects. We will therefore use `bottle`'s own template engine, which allows you to write normal `HTML` documents, which you can augment with snippets of python code. You can read about the templating in the `bottle` documentation: <https://bottlepy.org/docs/dev/tutorial.html#templates>.

From the `index` function, pass the `data` you queried from the `database` to the template function. In the `Index_Skeleton.tpl` file, create a `HTML` table. This should employ columns for each `data` field you queried (`Title`, `Subtitle`, etc).

Inject python code with the appropriate syntax, which loops over the queried `data` and fills the table. The `Archive` field should be a link, which leads you the archive's website. Run your server, visit its `URL` and check if everything works.

3. Augment your `HTML` table by adding one more column called `Thumbnail`. This should display a small version of the stored in each `data` entry. For this refer to the following tutorial: https://www.w3schools.com/tags/tag_img.asp.

Set the `thumbnail` to an appropriate size (e.g. 200 pixels). As source use the `/imageraw` route described above. Make sure you specify the correct `id` for each entry.

Test your website and enjoy it!

Now we will augment our `web server` by another route, which displays detailed information for a single `image` entry. As a reminder: The code skeleton is available on StudOn together with this assignment sheet or in the Kirmes repository. Just pull the latest version of the repo!

Problem 10.6 (Details Page)

Our overview table is nice, but we would like the user to be able to inspect a certain entry more closely. We will therefore create a new route, which displays information for a single `image` on its own page.

1. In your `Server.py` file, create a new route `/details/<id:int>`. Given an `Id` as argument, the function should **query** the `database` for this entry. If no entry with the `Id` can be found, use `bottle`'s `abort` function to display an error with the code 404: <https://bottlepy.org/docs/dev/tutorial.html#http-errors-and-redirects>.
2. Create a new template file `Details.tpl`. From your python code, call the template with the information you queried from the `database`. In the template, write `HTML` code which displays the given information in a nice and easy-to-read way.

Some information might not be available (`NULL/None`). Handle this case!

Test your page by navigating to the details `URL` for some example `image`, e.g. `http://localhost:8080/details/27`. Make sure, that all `data` is displayed correctly.

3. On the details page, also display the `image` in full size. You may again use the `/imageraw/id` route from last week as source.
4. Amend your `Index.tpl` from last week in the following way: Each `image thumbnail` in the table should be a link (``), which leads to the details page of this respective entry, i.e. by clicking on the `thumbnail` of `image 27` your website should navigate to the URL `http://localhost:8080/details/27`.

Problem 10.7 (New Entries and Editing)

The next step to creating a useful `web application` is to allow the user to insert new entries and edit existing ones.

We have prepared the code for adding new entries for you in this week's `Server.py` skeleton. If you want to continue with your own code, you can copy the functions `new`, `submitNew` and `getValue` from the skeleton to your own file. Also copy the file `New.tpl` to your directory. In your `Index.tpl`, add a link at the top of the page, which leads to the `/new` route.

Familiarize yourself with the given code. Understand how it works and how the `data` flows.

Editing entries is similar to adding new ones. Both require a form to insert `data`, which is then sent to a routine to handle the `database` calls. For the form the only difference is that some `data` is already filled out. For now we will only allow editing of the `metadata`, not the `digital image` itself. Your edit form does not need to allow changing the `digital image`.

1. Create a new file `Edit.tpl`. Take the given `New.tpl` as a starting point. Since we do not want to allow changing the `image` for now, you can omit the `Image` input field.
2. In your python code, create a new route `/edit/<id:int>`. In the function, `query` the `database` for the entry with the given `id`. Since this is the same operation as in the `/details/` route, you can reuse this code. Call the `Edit.tpl` template with your queried `data`.
3. For fields, which are already filled out, the form should display the current value. To this end, refer to the `value` attribute of the `<input>` fields. Test your page by navigating to the URL of an example entry, e.g. `http://localhost:8080/edit/27`. Make sure the available `data` is displayed correctly.
4. The key difference to the `New.tpl` form is, that we already have an entry, i.e. an `Id`. This must be passed via the form to the function, which handles the `database` update.

HTML forms allow hidden fields, which look like this:

```
<input type='hidden' name='id' value='{{id}}'>
```

Since the field is set to `hidden`, it will not show up on the web page. Nevertheless, its value (the `id`) will be sent with the rest of the filled out form `data`. Use the above code to add the `Id` to the form.

5. Create another route `/submit_edit` of type `POST`. Refer to the given `/submit_new` route for details. Obtain all `data` from the input form. Afterwards, issue an `SQL UPDATE` command to update the entry with the given `Id` and provide the values from the form.

In the end, use `bottle`'s `redirect` functionality to navigate to the details page of the edited entry. Again, refer to the `submitNew` function for details.

6. In the `Edit.tpl` file, make sure that the form action is set to the correct route.
7. On the details page, create a link `Edit`, which leads to your `/edit/<id>` route.

Chapter 10

Project: A Web GUI for a Books Database

In this chapter we will pull together the technologies we have learned into a simple [web application](#) project. We will do so in multiple setups. We first make a bare-bones application (see section 10.1) and then step by step extend it with new features:

- Ajax techniques for selectively loading page fragments: section 10.3
- Access control and management: section 10.2
- Deploying Python applications as programs: section 10.4

Bricolage Programming: With this project we want to demonstrate a common practice of modern [programming](#): pulling together [program](#) fragments or solution ideas from various sources (e.g. the IWGS course notes or various tutorials or even answers from stack overflow <https://stackoverflow.com>, a question and answer site for professional and enthusiast [programmers](#)) and then adapting them to the current project and fitting them together into a coherent [program](#) that works as expected.

This approach to [programming](#) is often called “[bricoleur style](#)” [Tur95] or [bricolage programming](#) because it relies on handicraft-like tinkering with pieces of existing materials.

Contrary to what many classical [programming](#) courses still insinuate they seem to say that you have to know everything before you can start with a project – the advent of the internet with its multitude of high-quality [programming](#) related resources has made [bricoleur style programming](#) [effective](#) and [efficient](#).

Actually, [bricolage](#) is a technique that should be leaned and adopted as a tool, especially for part-time [programmers](#) as practitioners in the digital humanities tend to be.

The [web application](#) project in this chapter is a [bricolage](#) project, only that we have almost all the ideas in the IWGS course notes already and we do not have to google for them on the web.

10.1 A Basic Web Application

We bring together all we have learned into a basic [web application](#) that allows to list all the books in a [database](#), as well as add, edit, and delete book records.

We use our running example of the books table as a basis, and add a [web application](#) layer via the [bottle WSGI server-side scripting framework](#) in [Python](#).

We have intentionally kept the application very simple, so that it can serve as the basis of other projects. The full source is available at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/databases/code/books-app.py>. The respective template files are siblings.

Building a full Web Application with Database Backend

- ▷ **Observation 10.1.1.** *With the technology in chapter 5 (Web Applications) in the IWGS lecture notes and chapter 9 we can build a full [web application](#) in less than*
 - ▷ 100 lines of [Python](#) code and *(back-end/routes)*
 - ▷ less than 70 lines of [HTML](#) template files. *(front end)*
- ▷ **Functionality:** Manage a [database](#) of books, in particular: *(e.g. your library at home)*
 - ▷ add a new book to the [database](#)
 - ▷ delete a book from the [database](#)
 - ▷ update (i.e. change) an existing book
- ▷ The source is at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/booksapp/code/books-app.py>.

Now, if you download the file `books-app.py` and all the sibling template files `*.tpl` at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/booksapp/code/>, you can start the application from the terminal by typing `python3 books-app.py`. This will yield something like

```
> python3 books-app.py
Bottle v0.12.18 server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

So enter the url `http://localhost:8080/` into the [URL](#) bar of your browser, and test the setup. But let us return to the [implementation](#) of the [web application](#).

We do the usual things to set up the [web application](#): we load the libraries, connect to the data base, and so on.

The Books Application: Setup

- ▷ We have already seen how to set up the [database](#) in slide 248.

```
import sqlite3
from sqlite3 import Error
from bottle import route, run, debug, template, request, get, post

# our database file
database = "books.db"
db = sqlite3.connect(database)
```

- ▷ But we want to receive result rows as dictionaries, not as tuples, so we add


```
db.row_factory = sqlite3.Row
```
- ▷ We give ourselves a cursor to work with


```
cursor = db.cursor()
```
- ▷ We start the bottle server

```
run(host='localhost', port=8080, debug=True)
```

- ▷ And of course, we eventually commit and close the `database` in the end

```
db.commit()
db.close()
```

The next step is to create a table for the books. This is a completely standard `SQL CREATE` statement which we execute in the cursor we have established during setup.

The Books Application: Backend

- ▷ We specify the `database schema` and create the Books table

```
bookstable = """
CREATE TABLE IF NOT EXISTS Books (
    Last varchar(128), First varchar(128),
    YOB int, YOD int, Title varchar(255), YOP int,
    Publisher varchar(128), City varchar(128)
);
"""
cursor.execute(bookstable)
```

The next step is strictly optional. But it is so annoying to have to start with an empty `database` when the `web application` first comes up. So we provide a list of seven books. But, if we are not careful, these books will be inserted into the `database` every time we start up the application. Recall that we did not drop the Books table in the code above.

The Books Application: Books to Play With

- ▷ Data about books as a `Python` list of 8-tuples:

```
initialbooklist = [
    ('Twain', 'Mark', 1835, 1910, 'Huckleberry_Finn', 1986, 'Penguin_USA', 'NY'),
    ('Twain', 'Mark', 1835, 1910, 'Tom_Sawyer', 1987, 'Viking', 'NY'),
    ('Cather', 'Willa', 1873, 1947, 'My_Antonia', 1995, 'Library_of_America', 'NY'),
    ('Hemingway', 'Ernest', 1899, 1961, 'The_Sun_Also_Rises', 1995, 'Scribner', 'NY'),
    ('Wolfe', 'Thomas', 1900, 1938, 'Look_Homeward_Angel', 1995, 'Scribner', 'NY'),
    ('Faulkner', 'William', 1897, 1962, 'The_Sound_and_the_Fury', 1990, 'Random_House', 'NY'),
    ('Tolkien', 'John_Ronald_Reuel', 1892, 1973, 'The_Hobbit', 1937, 'George_Allen_Unwin', 'UK')]
```

- ▷ If the Books table is empty, we fill it with the tuples in `initialbooklist`:

```
row = cursor.execute('SELECT * FROM Books LIMIT 1').fetchall()
if not row:
    cursor.executemany('INSERT INTO Books VALUES(?,?,?,?,?,?,?)', initialbooklist)
```

- ▷ **Idea:** To find out if the table is empty (surprisingly clumsy)
 - ▷ we fetch a list with at most one row (`LIMIT 1`);
 - ▷ if Books is empty, `row` is the empty list which evaluates to false in a conditional.

In a more complete version of the books application we would probably have used a keyword argument like `--initbooks` to the program. We will cover command line [parsing](#) – the technology that enables behavior modifiers – section 10.4. The next task is to create a route for the main page of the application, i.e. the page `booksapp.py` serves at `http://localhost:8080/`. We want a listing of all the books in the [database](#) in a table.

The Books Application Routes: The Application Root

- ▷ We only need to add the [bottle routes](#) for the various sub pages.
- ▷ **The main page:** Listing the book records in the [database](#)

```
@route('/')
def books():
    query = 'SELECT rowid, Last, First, YOB, YOD, Title, YOP, Publisher, City FROM Books'
    cursor.execute(query)
    booklist = cursor.fetchall()
    return template('books', books=booklist, num=len(booklist), cols=cols)
```

- ▷ This uses the following templates: the first generates a table of books from the template file `books.tpl`

```
<p>There are {{num}} books in the database</p>
<table>
    % include('th.tpl', cols=cols)
    % for book in books : include('book.tpl', **book, cols=cols) end
    <tr><th><a href="/add"><button>add a book</button></a></th></tr>
</table>
```

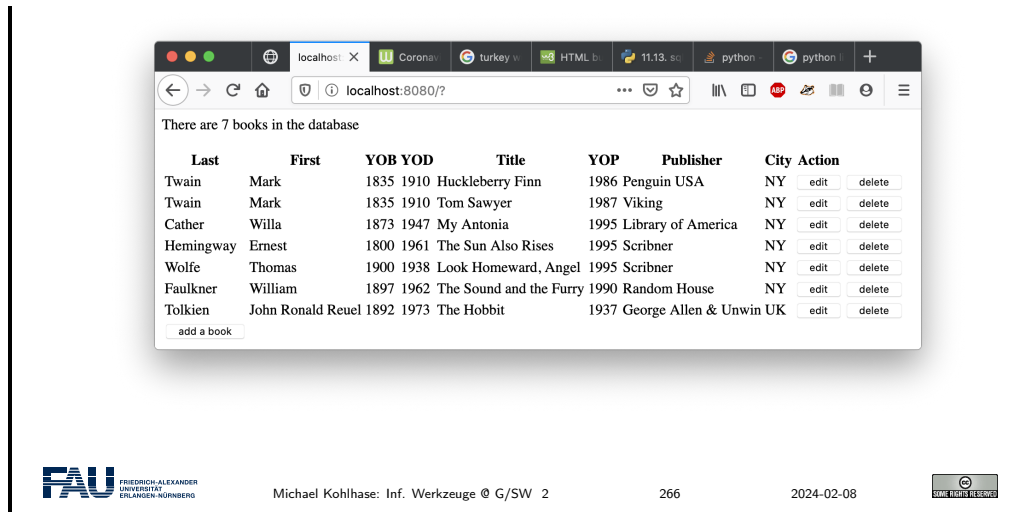
The backend of this is very simple: we fire up a simple [SQL query](#) that selects all the records from the `Books` table. As we configured the [database](#) connection to return [database](#) records as [Python](#) dictionaries, the variable `booklist` variable is a list of data dictionaries, which we can feed to the STPL template `books.tpl`, which creates the return page for `http://localhost:8080/`. This page consists of a paragraph which reports on the number of books in the [database](#) and then a table which is built up from

1. a table header which is simply imported from a template file `th.tpl`
2. a body, which is created by iterating over `booklist`, feeding each row – a [Python](#) dictionary – to the template `book.tpl` as [keyword arguments](#) via the [double star operator](#), and
3. a table row with a link to the `add` route for adding new books.

Before we show the nested templates, let us inspect the result:

The Books Application Root: Result

- ▷ Here is the page of the books application in its initial state.



Indeed we have the report on the number of books and a table which ends in an “add a book” link. The table header and rows contain the seven data cells and two more for possible actions on the [database](#) records. The next two templates are responsible for that; they are called in the books template above.

The Books Application Root: More Templates

- ▷ **Recall:** The books.tpl template file

```
<p>There are {{num}} books in the database</p>
<table>
  % include('th.tpl', cols=cols)
  % for book in books : include('book.tpl',**book,cols=cols) end
  <tr><th><a href="/add"><button>add a book</button></a></th></tr>
</table>
```

that generates this result via the following two templates:

- ▷ It inserts the table header via th.tpl:

```
% for col in cols:
  <th>{{col}}</th>
% end
<th rowspan="2">Action</th>
```

- ▷ and iterates over the list of books, using the template file book.tpl:

```
<tr>
  <td>{{Last}}</td><td>{{First}}</td><td>{{YOB}}</td><td>{{YOD}}</td>
  <td>{{Title}}</td><td>{{YOP}}</td><td>{{Publisher}}</td><td>{{City}}</td>
  <td><a href="/edit/{{rowid}}"><button>edit</button></a></td>
  <td><a href="/delete/{{rowid}}"><button>delete</button></a></td>
</tr>
```

- ▷ **Row Id Trick:** Note the slightly subtle use of the rowid column in this template. It is (only) used in the two action buttons to specify which book to add/edit.

The template th.tpl is completely elementary, book.tpl is called with [keyword arguments](#) whose values substituted for the `{{key}}` template variables. The last two columns in the table are the action links that point to the add and delete routes we present next. The “add a book” functionality is distributed over two routes: a GET route for the path `/add/`

Note the use of sqlite3 [parameter substitution](#) in addResponse!

The addResponse function that answers the POST route for the path /add/ just inserts a new [database](#) record in to the Books table. Note the use of the SQLite3 [parameter substitution](#) here. We substitute the [parameters](#) :«key» in the string ins with the corresponding values in the [Python](#) dictionary data which we obtain as the result of the parseResponse function, which we will look at next.

The Books Application Routes: Adding Book Records

- ▷ This uses the function parseResponse, which we will reuse later.

```
def parseResponse ():
    data = {'Last': request.forms.get('Last'),
           'First': request.forms.get('First'),
           'YOB': request.forms.get('YOB'),
           'YOD': request.forms.get('YOD'),
           'Title': request.forms.get('Title'),
           'YOP': request.forms.get('YOP'),
           'Publisher': request.forms.get('Publisher'),
           'City': request.forms.get('City')}
    return data
```

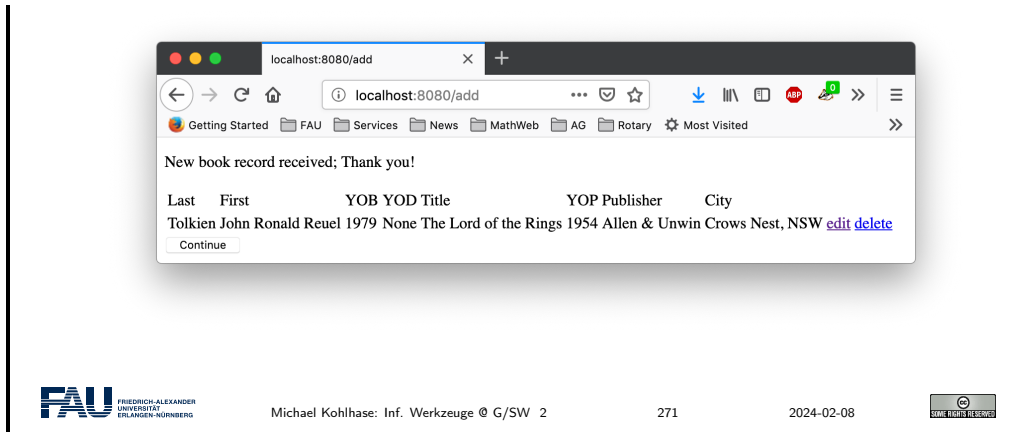
- ▷ and the template response.tpl:

```
<form action="/">
  <p>{{text}}; Thank you!</p>
  <table>
    % include('th.tpl',cols=cols)
    % include('book.tpl',**data,cols=cols)
  </table>
  <input type="submit" value="Continue" />
</form>
```

The parseResponse function is almost trivial, it just [queries](#) the response object that comes from the POST request for the various components via the forms.get method and packages the results in a [Python](#) dictionary that feeds the response.tpl template. The latter creates a [HTML](#) form without text input fields we only use it to trigger a GET request to the path / (the application root that displays the updated book list). Note that we re-use the templates th.tpl and books.tpl from above again.

The Books Application Routes: Adding Book Records

- ▷ Here is the result after filling in Tolkien's "Lord of the Rings":



The next relevant route is the “delete a book” functionality. Here we use another new feature: when creating a `database` table in SQLite3, the system creates an additional primary key column `rowid`. In particular we have a `rowid` column in the `Books` table, which we make use of.

The Books Application Routes: Deleting Book Records

- ▷ We add a route for deleting book records (for the delete button)

```
@get('/delete/<id:int>')
def delete(id):
    cursor.execute('DELETE FROM Books WHERE rowid = ?', (id,))
    return template('delete')
```

Note that we have a `dynamic route` here: We use the `named wildcard` `<id:int>` to obtain the `rowid` of the record to be deleted.

- ▷ The template file `delete.tpl` does the obvious:

```
<form action='/'>
  <p>Book record deleted; Thank you!</p>
  <input type="submit" value="Continue"/>
</form>
```

Note that the link on the “delete” buttons in the books table root (see template `book.tpl` above) has the form `<button href="/edit/{rowid}">edit</button>`, i.e. it references the `rowid` column. This is picked up in the GET route for `/delete/<id:int>` path via the `named wildcard` `<id:int>`. This makes sure the right `database` record is deleted.

The routes for editing book records combine techniques from the ones for adding and deleting. From the former we use the layout into a GET and POST route, from the latter, we use the `dynamic route`

The Books Application Routes: Editing Book Records

- ▷ **Idea:** Combine techniques from the add and delete routes

```
@get('/edit/<id:int>')
def edit(id):
    cursor.execute('SELECT * FROM Books WHERE rowid = ?', (id,))
```

```

return template('edit',cursor.fetchone(),id = id,cols=cols)

@post('/edit/<id:int>')
def editResponse(id):
    data = parseResponse()
    up = """UPDATE Books
           SET Last = :Last, First = :First, YOB = :YOB, YOD = :YOD,
             Title = :Title, YOP = :YOP, Publisher = :Publisher,
             City = :City
           WHERE rowid = :rowid"""
    data.update({'rowid': id})
    cursor.execute(up,data)
    return template('response',data=data,text='Updated book record',cols=cols)

```

In this case we have a small subtlety: the update instruction and the template edit.tpl need a rowid key/value pair. We solve this by updating the data dictionary suitably. Now we only have to give the template edit.tpl, which is rather straightforward.

Books Application Routes: Editing Book Records (cont.)

- ▷ The template file edit.tpl is similar to add.tpl above, but pre-fills the input fields with the `database` record values.

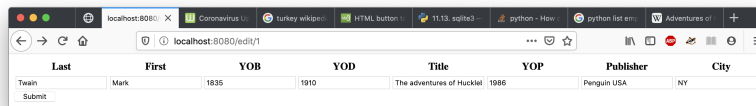
```

<form action="/edit/{{id}}" method="post">
  <table>
    % include('th.tpl', cols=cols)
    <tr>
      <td><input type="text" name="Last" value="{{Last}}"/></td>
      <td><input type="text" name="First" value="{{First}}"/></td>
      <td><input type="text" name="YOB" value="{{YOB}}"/></td>
      <td><input type="text" name="YOD" value="{{YOD}}"/></td>
      <td><input type="text" name="Title" value="{{Title}}"/></td>
      <td><input type="text" name="YOP" value="{{YOP}}"/></td>
      <td><input type="text" name="Publisher" value="{{Publisher}}"/></td>
      <td><input type="text" name="City" value="{{City}}"/></td>
      <td><input type="submit" value="Submit"/></td>
    </tr>
  </table>
</form>

```

Books Application Routes: Editing Book Records (cont.)

- ▷ The result is



▷ Again, we use the template `response.tpl`, which we fill with a different message.

The main message to take home from this experiment is that we can build a simple but complete [web application](#) with less than 100 lines of [Python](#) code and less than 70 lines of [HTML template files](#).

10.2 Access Control and Management

Now that we have a basic [web application](#) running, we can start adding features. The most important one is [access control](#) to restrict who can access more critical functionalities of the [web application](#), such as deleting or editing [database](#) entries.

There are many technologies for [access control](#), many use advanced features like browser [cookies](#). Here we want to introduce the simplest one: [HTTP basic authentication](#) is built into the fabric of the world wide web: it is part of the [HTTP](#) protocol that drives it.

As [HTTP basic authentication](#) is unsafe (it sends user names and passwords over the network only lightly encoded), we also add a discussion on how to upgrade the [web application](#) to [HTTPS](#).

The full source is available at <https://gl.mathhub.info/MiKoMH/IWGS/blob/master/source/databases/code/books-app-https.py>. The respective template files are siblings.

Access Control and Management

- ▷ **Problem:** Anyone can write, edit, and delete records from the books [database](#).
- ▷ **Solution:** Implement a password-based [log in](#) procedure and restrict write/edit/delete access to logged-in agents.
- ▷ Let's fix some terminology before we continue
- ▷ **Definition 10.2.1.** [Access control](#) is the selective restriction of access to a resource, [access management](#) describes the corresponding process.
- ▷ [Access management](#) usually comprises both [authentication](#) and [authorization](#).
- ▷ **Definition 10.2.2.** [Authorization](#) refers to a set of rules that determine who is allowed to do what with a collection of resources.
- ▷ **For our books application** we need four things
 1. a browser [interaction](#) to query the user for username and password
 2. a way to transport them to the [web application](#) program
 3. a method for checking the username/password ([authentication](#))
 4. a way the specify who can do what. ([authorization](#))

Realization: 1./2. via [HTTP](#), 4. via [bottle basic auth](#), [implement](#) 3. directly.

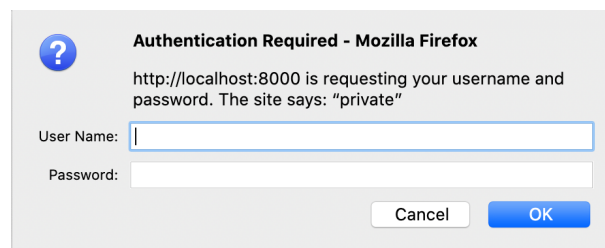
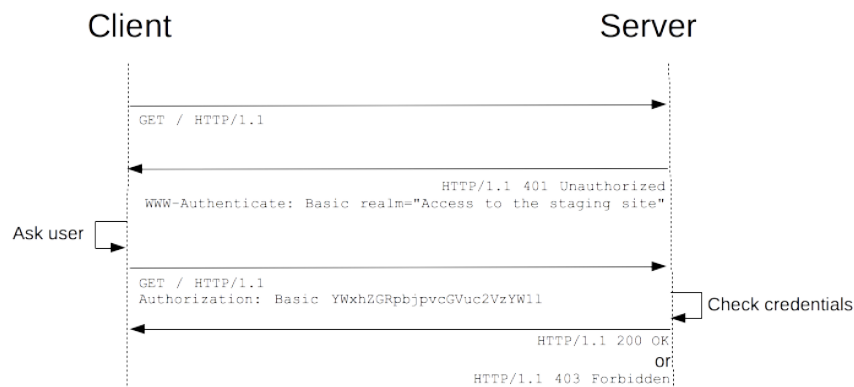
[HTTP basic authentication](#) is a simple mechanism in the [HTTP](#) protocol that standardizes the transmission of username/password information the “handshake” that leads to its acquisition.

HTTP Basic Authentication

- ▷ **Recall** that **HTTP** is a plain text protocol that passes around headers like this

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
```

- ▷ **Idea:** For **authentication** extend the **HTTP** headers with support for username/-password pairs.
- ▷ **Definition 10.2.3.** **HTTP basic authentication** introduces a **HTTP** header **Authorization** for **base64 encoded** pairs `«username»:«password»` and a couple of challenge/response messages.



- ▷ **Problem:** **Base64** is very easy to decode, so usernames and passwords are communicated in the clear (very unsafe)
- ▷ Passwords are “binary data” (think special characters), encoding just keeps them unchanged over the network. (no encryption)

The message sequence diagram in Definition 10.2.3 shows the basic handshake mechanism that establishes **authentication** and the delivery of restricted material to an **authenticated** user.

The diagram shows the details of the communication between client and server (symbolized by the two vertical lines). The top arrow is a normal **HTTP GET** request (without a **Authorization**

field).

But – as the resource that is requested is access-restricted – the server does not just answer with a [HTTP](#) “200 OK” and the resource, instead the server answers with a [HTTP](#) “401 Unauthorized” code, which contains a description of the reason for the restriction.

When the [browser](#) receives the 401 code, it asks the user for a user name and password e.g. with a [popup](#) form like the one shown in Definition 10.2.3, possibly displaying the reason string – here “private”. This information is then sent to the server in a second [GET](#) request, this time with the username/password information in the [Authorization](#) request.

The server checks the user/password data and – depending on the result of the check – sends a [HTTP](#) response “200 OK” together with the resource or a “403 Forbidden” (without the resource).

⚠ One thing that we have not discussed here is that most [browsers](#) store the username/password information and supply it to the server – often directly in any outgoing requests – which makes it hard to test [authentication](#) and [unauthenticated](#) behavior in [web application](#) development. A useful trick here is – if you are logged into `http://example.org` – to address a [GET](#) request to `http://abc@example.org`. Background: [HTTP basic authentication](#) allows you to set user/password information directly by prepending `<<user>>:<<pass>>` to the [authority](#) of the [URI](#) used in a [HTTP](#) request.

Of course, [HTTP basic authentication](#) is supported by the [bottle WSGI](#) framework.

Basic Auth in Bottle

- ▷ **Idea:** Support the server side of [HTTP basic authentication](#) in [bottle web-apps](#).
- ▷ **Implementation:** New decorator `@auth_basic(⟨⟨function⟩⟩)` to mark a [route](#) as password-protected.
- ▷ **Usage:** Decorate every route we want to restrict access of with `@auth_basic(⟨⟨function⟩⟩)`, where `⟨⟨function⟩⟩` is a function that takes two string arguments (user name and password) and returns a Boolean for the [authorization](#) decision.

What happens behind the scene here is clear from the [authentication](#) handshake explained in Definition 10.2.3

Basic Auth in Bottle: Minimal Viable Example

- ▷ **Example 10.2.4.** A [web application](#) with restricted route.

```
from bottle import run, get, auth_basic

def check(user, password):
    return user == "miko" and password == "test"

@get("/")
@auth_basic(check)
def protected():
    return "Authorized␣access␣granted!"

run(host="localhost", port=8000)
```

- ▷ **Idea:** Mix restricted and open routes in a partially restricted application.

- ▷ **Extension:** Use different check functions for different levels of restriction (user roles)

This was easy enough. But one problem remains: in [HTTP basic authentication](#), user names and passwords are not confidential when they are transported over the network. The simplest way to ensure confidentiality is to layer encryption on top of [HTTP](#), which is just what the [HTTPS](#) protocol does.

HTTPS: HTTP over TLS

- ▷ **Definition 10.2.5.** [Hypertext Transfer Protocol Secure \(HTTPS\)](#) is an extension of the [Hypertext Transfer Protocol \(HTTP\)](#) for secure communication over a [computer network](#). [HTTPS](#) achieves this by running [HTTP](#) over a [TLS](#) connection.
- ▷ **Consequences for Web Applications:** We can use [HTTP](#) as usual, except
 - ▷ we gain communication privacy and server [authentication](#),
 - ▷ server and browser need to speak [HTTPS](#), (most do)
 - ▷ the server needs a [public key certificate](#) and a [private key](#).
- ▷ In bottle, we can just swap out the [HTTP](#) server to one that can do [HTTPS](#):


```
run(host='localhost',port='8888',
    server='gunicorn',keyfile='key.pem',certfile='cert.pem')
```

install it first with pip install gunicorn.
- ▷ **Problem:** Where to get the certificate file cert.pem and private key key.pem?
- ▷ **One Solution:** Self-sign one, e.g. using <https://www.selfsignedcertificate.com/> (adapt file names)
- ▷ **Remaining Problem:** Your browser forces you to specify an exception for `https://localhost:8888` (probably OK for development)

Self-signed [TLS](#) certificate are sufficient for [web application](#) development. But publically deploying a [HTTPS](#) based web application we need real ones. Fortunately, there is a relatively simple way of obtaining them.

Getting a Real TLS Certificate via Let's-Encrypt

- ▷ **Intuition:** [HTTPS](#) is the new "regular [HTTP](#)" on the web!
- ▷ **Observation 10.2.6.** A self-signed certificate gives communication privacy but not [authentication](#) ↔ only you yourself vouch for the authenticity of the [web site](#).
- ▷ **Definition 10.2.7.** In a [public key infrastructure](#), the [TLS](#) certificate is issued by a [certificate authority](#), an organization chartered to verify identity and issue [TLS](#) certificates.

- ▷ Commercial [certificate authorities](#) sell trust. (for a lot of money)
They certify e.g. that the `https://bmw.com` is under control of BMW AG.
- ▷ **Idea:** Finding out that you have control over a particular [web site](#) on the [web](#) can be automated, if you run a program on the server host.
- ▷ **Definition 10.2.8.** [Let's Encrypt](#) is a not for profit [certificate authority](#) that does this and issues [free TLS](#) certificates. (to encourage HTTPS adoption)
- ▷ **Concretely:** on a linux server you need two steps
 1. `install` certbot (usually via your package manager)
 2. then `sudo /usr/local/bin/certbot certonly --standalone` will generate certs.


Details at <https://letsencrypt.org>.
- ▷ **Success:** $\geq 1.000.000.000$ TLS certificates, 200.000.000 sites since 2016

We have only covered the basic ideas behind certificate authorities and [Let's Encrypt](#) here, but this should enable you to figure out the rest from the [Let's Encrypt web site](#).

10.3 Asynchronous Loading in Modern Web Apps

The [web applications](#) we have seen up to now have been relatively conventional, based mostly on server-side scripting together with some client-side computation via [JavaScript](#). This is a powerful setup with one problem. Whenever the user needs new data from the server, the browser has to request a new [web page](#) – even if only a small fragment of the original page needs to be changed.

The solution to this problem is to use [JavaScript](#) itself to load the new information and directly integrate the result into the [DOM](#), using a technology called [Ajax](#). In this section we will introduce [Ajax](#) by extending the [database](#) from section 10.1 with a lightweight front-end [web application](#).

Before we get into the example, we introduce [Ajax](#) as a technology itself and recap the idea of client-side computation using the [DOM](#).  The code in this section will be considerably more complex than what we have seen before. But it shows many of the characteristic ideas of modern web application development in a nutshell. That should make it worthwhile to study, even if that may take more than one attempt.

AJAX for more responsive Web Pages

- ▷ **Definition 10.3.1.** [Ajax](#), (also [AJAX](#); short for “Asynchronous [JavaScript](#) and [XML](#)”) is a set of client side techniques for creating [asynchronous web applications](#).
- ▷ **Definition 10.3.2.** A [process](#) p is called [asynchronous](#), iff the parent process (i.e. the one that spawned p) continues processing without waiting for p to terminate.
- ▷ **Intuition:** With [Ajax](#), [web applications](#) can send and retrieve data from a [server](#) without interfering with the display and behaviour of the existing page.
- ▷ **Application:** By decoupling the data interchange layer from the presentation layer, [Ajax](#) allows [web pages](#) and, by extension, [web applications](#), to change content dynamically without the need to reload the entire [page](#).
- ▷ **Observation:** Almost all modern [web application](#) extensively utilize [Ajax](#).

- ▷ **Note:** In practice, modern implementations commonly use JSON instead of XML.

Recall the HTML rendering pipeline in browsers around the DOM we introduced for client-side computation.

Background: Rendering Pipeline in browsers

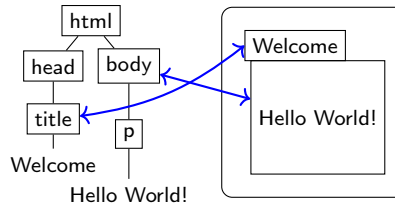
- ▷ **Observation:** The nested markup codes turn HTML documents into trees.
- ▷ **Definition 10.3.3.** The **document object model (DOM)** is a **data structure** for the HTML document tree together with a standardized set of access methods.
- ▷ **Rendering Pipeline:** Rendering a web page proceeds in three steps
 1. the browser receives a HTML document,
 2. parses it into an internal data structure, the DOM,
 3. which is then painted to the screen. (repaint whenever DOM changes)

HTML Document

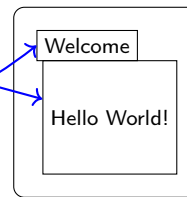
```
<html>
<head>
  <title>Welcome</title>
</head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

parse →

DOM



Browser



The DOM is notified of any user events

(resizing, clicks, hover, ...)

The most important concept to grasp here is the tight synchronization between the DOM and the screen. The DOM is first established by parsing (i.e. interpreting) the input, and is synchronized with the browser UI and document viewport. As the DOM is persistent and synchronized, any change in the DOM is directly mirrored in the browser viewpoint, as a consequence we only need to change the DOM to change its presentation in the browser. This exactly is the purpose of the client side scripting language, which we will go into next.

We will put the abstract ideas about Ajax and JSON introduced above to practical use. This will make our understanding much more concrete.

The first step in the development of a Ajax based front end for the books application – as in any software project – is to specify the intended behaviour of the front-end and plan the implementations.

Example: Details on Request via AJAX

- ▷ **Idea:** Use Ajax in a web application for the books application
 - ▷ The start page just has a list of book titles, and
 - ▷ details are fetched by an Ajax request and presented in line.

- ▷ **Planning the Program:** We need a bottle `server` with
1. a `dynamic route` that returns `JSON`-encoded data for a given book,
 2. a `route` for the main page that lists the book titles,
 3. `stpl template files` for list items with an `Ajax` request, and
 4. a `JavaScript` function that reads the `JSON` and inserts it into the `DOM`.

Here we see a mockup of what the result will look like:

The finished product (initial state)

Books by Title

1. Tom Sawyer (**show details**)
2. My Antonia (show details)
3. The Sun Also Rises (show details)
4. Look Homeward, Angel (show details)
5. The Sound and the Furry (show details)
6. The Hobbit (show details)

The finished product (with details loaded)

Books by Title

1. Tom Sawyer
Author: Mark Twain (1835 - 1910)
Publisher: Viking, 1987
(hide details)
2. My Antonia (show details)
3. The Sun Also Rises (show details)
4. Look Homeward, Angel (show details)
5. The Sound and the Furry (show details)
6. The Hobbit (show details)

Now we are ready to begin with the `implementation`. Fortunately, the first step – serving the main page and the `JSON` data for a given book is very simple, indeed that is exactly what bottle was created for, since it is a routine task for building modern `web applications`.

The Routes (Serving HTML and JSON)

- ▷ After setting up the `database` and `co`, we have a standard route:

```
@route('/')
def books():
    cursor.execute('SELECT rowid, Title, YoP FROM Books')
    rv = cursor.fetchall()
    return template('titles', books=rv)
```

- ▷ **JSON** routes and APIs are very easy in bottle: we just return a dictionary.

```
@route('/json/<id:int>')
def book(id):
    cursor.execute(f'SELECT * FROM Books WHERE rowid={id}')
    row = cursor.fetchone() # Only one result, rowid is a primary key.
    return dict(zip(row.keys(), row)) # Pair up column names with values.
```

- ▷ **Dictionaries and JSON in Bottle:** Bottle automatically transforms **Python** dictionaries into **JSON** strings; sets the Content Type header to `application/json`.

The Basic Templates

- ▷ The template `titles.tpl` is also standard

```
<html>
% include('bookshead.tpl')
<body>
  <h1>Books by Title</h1>
  <ol>
    % for bk in books: include('title.tpl', id=bk[0], title=bk[1]) end
  </ol>
</body>
</html>
```

- ▷ The template `title.tpl` presents a single book title

```
<li>
  <span class="booktitle">{{title}}</span>
  <span id="content{{id}}"></span>
  <span class="interact" id="interact{{id}}"
    onclick="load_details('{{id}})">(show details)</span>
</li>
```

The empty span will be filled by an **Ajax** call later!

- ▷ The interesting things happen in `bookshead.tpl`

(up next)

But now it becomes more tricky. We set up a couple of scripts in head of `bookshead.tpl`, which we will now take a more detailed look at.

The Script load_details

- ▷ bookshead.tpl starts supplying **JQuery** and a **JQuery** templating library:

```
<script type="application/javascript"
  src="http://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
<script type="application/javascript"
  src="https://cdn.jsdelivr.net/gh/codepb/jquery-template@1.5.10/dist/jquery.loadTemplate.min.js"></script>
```

- ▷ The main contribution of bookshead.tpl is the **JQuery** function load_details

```
async function load_details (numb) {
  /* Request Info via JSON, feed it to template, update "show_details" span */
  await $.getJSON("/json/" + numb,
    function (data) {$("#content" + numb).loadTemplate($("#open"), data)});
}
```

which uses the **JQuery Ajax** call \$.getJSON. This takes two arguments:

1. the **URL** for the HTTP GET request
2. a **JavaScript** function that is called if the GET request was successful.

The **function** (in **argument** 2) is then used to extend the result of \$("#content" + numb), i.e. that element in the **DOM** whose id attribute is content_i where *i* is the value of the numb variable.

The Script load_details Continued

- ▷ We also use **JQuery** to change the onclick behaviour of the span element (from load_details to toggle_details, explained below) and the text contained therein.

```
interact = $("#interact" + numb)
/* change click behaviour of interaction span from show to toggle */
interact.removeAttr('onclick');
interact.attr('onClick', 'toggle_details(' + numb + ');');
/* also change included text appropriately */
interact.html("hide_details");
}
```

- ▷ Recall the structure of title.tpl: For every book we have a title, a content element that starts out empty and gets filled when load_details is called, and a clickable **interaction** element that triggers load_details.

```
<li>
  <span class="booktitle">{{title}}</span>
  <span id="content{{id}}"></span>
  <span class="interact" id="interact{{id}}"
    onclick="load_details({{id}}">(show details)</span>
</li>
```

- ▷ The toggle_details-function used above does nothing but setting the content element to hidden or visible and changing the text of the **interaction** element.

```
function toggle_details (numb) {
  /* hide or show appropriate content element */

  content = $("#content" + numb);
  interact = $("#interact" + numb);

  if(content.css('display') === 'none') {
    content.show();
    interact.html("hide_details");
  } else {
    content.hide();
    interact.html("show_details");
  }
}
```

Now let us look at this process in more detail. Apart from the fact that we are using [jQuery template processing](#) and the syntax is different, this works exactly like [bottle template processing](#), which we have extensively practiced above. So just buckle up and enjoy the ride.

jQuery Template Processing

- ▷ **Recall:** We are still trying to understand
`$("#content" + numb).loadTemplate($("#open"),data)`
 It extends the empty `` in `title.tpl` with a details table:
- ▷ The `loadTemplate` method takes two arguments

1. a template; here the result of `$("#open"),` i.e. the element in `bookshead.tpl` whose `id` attribute is `open` (note the type attribute that makes it HTML)

```
<script type="text/html" id="open">
  <table>
    <tr>
      <th>Author:</th>
      <td>
        <span data-content="First"></span> <span data-content="Last"></span>
        (<span data-content="YOB"></span> - <span data-content="YOD"></span>)
      </td>
    </tr>
    <tr>
      <th>Publisher:</th>
      <td><span data-content="Publisher"></span>, <span data-content="YOP"></span></td>
    </tr>
  </table>
</script>
```

2. a [JavaScript](#) data object: here the argument of the success function: the [JSON](#) record provided by the server under route `/json/i`

```
{ "Last": 'Twain',
  "First": 'Mark',
  "YoB": 1835,
  "YoD": 1910,
  "Title": 'Huckleberry_Finn',
  "YoP": 1986,
  "Publisher": 'Penguin_USA',
  "City": 'NY' }
```

- ▷ The [jQuery template processing](#) places the value of the data—content attribute into the ``. The resulting table constitutes the generated “detail view”:

```
<table>
<tr>
<th>Author:</th>
<td>
<span>Mark</span> <span>Twain</span>
(<span>1835</span>—<span>1910</span>)
</td>
</tr>
<tr>
<th>Publisher:</th>
<td><span>Penguin USA</span>, <span>NY</span></td>
</tr>
</table>
```

- ▷ **Note:** Both the [JavaScript](#) object in step 2, as well as the result of the [template processing](#) show afterwards are virtual objects that exist only in memory. In particular, we do not have to write them explicitly.

Now, we will show you the code in its entirety, it is less than 100 lines. So with the right tools, a modern [web page](#) with [Ajax](#) is not that difficult (once you wrap your head around it).

Code: An AJAX-based Frontend for the Books App

- ▷ `booksapp-ajax.py`: the [web server](#) with two routes

```
import sqlite3
from bottle import route, run, template, static_file

# Connect to database
db = sqlite3.connect("./books.db")
# Row factory so we can have column names as keys.
db.row_factory = sqlite3.Row
cursor = db.cursor()

@route('/')
def books():
    cursor.execute('SELECT rowid, Title, YoP FROM Books')
    rv = cursor.fetchall()
    return template('titles', books=rv)

# JSON interfaces are very easy in bottle, just return a dictionary
@route('/json/<id:int>')
def book(id):
    cursor.execute(f'SELECT * FROM Books WHERE rowid={id}')
    row = cursor.fetchone() # Only one result, rowid is a primary key.
    return dict(zip(row.keys(), row)) # Pair up column names with values.

run(host='0.0.0.0', port=32500, debug=True)
# Close database
db.close()
```

- ▷ `titles.tpl` styles the list of book titles

```
<html>
% include('bookshead.tpl')
<body>
<h1>Books by Title</h1>
<ol>
% for bk in books: include('title.tpl', Id=bk[0], title=bk[1]) end
</ol>
</body>
```



```
</html>
```

▷ title.tpl styles a single book

```
<li>
  <span class="booktitle">{{title}}</span>
  <span id="content{{id}}"></span>
  <span class="interact" id="interact{{id}}"
    onclick="load_details({{id}})">(show details)</span>
</li>
```

▷ bookshead.tpl provides the whole head of the main page.

```
<head>
  <title>Books with Ajax Details</title>
  <meta charset="utf-8">
  <style>.interact:hover { background-color: yellow; }</style>

  <script type="application/javascript"
    src="http://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
  <script type="application/javascript"
    src="https://cdn.jsdelivr.net/gh/codepb/jquery-template@1.5.10/dist/jquery.loadTemplate.min.js"></script>

  <script type="text/html" id="open">
    <table>
      <tr>
        <th>Author:</th>
        <td>
          <span data-content="First"></span> <span data-content="Last"></span>
          (<span data-content="YOB"></span> - <span data-content="YOD"></span>)
        </td>
      </tr>
      <tr>
        <th>Publisher:</th>
        <td><span data-content="Publisher"></span>, <span data-content="YOP"></span></td>
      </tr>
    </table>
  </script>

  <script type="text/javascript">
  /* async because we're waiting for the template magic to finish before appending */
  async function load_details (numb) {
    /* Request Info via JSON, feed it to template, update "show_details" span */
    await $.getJSON("/json/" + numb,
      function (data) {$("#content" + numb).loadTemplate($("#open"), data)});

    interact = $("#interact" + numb)

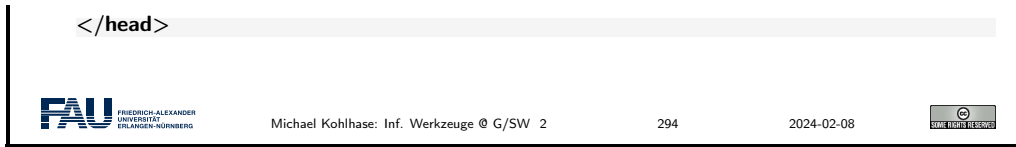
    /* change click behaviour of interaction span from show to toggle */
    interact.removeAttr('onclick');
    interact.attr('onClick', 'toggle_details(' + numb + ');');

    /* also change included text appropriately */
    interact.html("hide_details");
  }

  function toggle_details (numb) {
    /* hide or show appropriate content element */

    content = $("#content" + numb);
    interact = $("#interact" + numb);

    if(content.css('display') == 'none') {
      content.show();
      interact.html("hide_details");
    } else {
      content.hide();
      interact.html("show_details");
    }
  }
</script>
```



10.4 Deploying the Books Application as a Program

Now we address the fact that a web application is usually deployed on a unix server, by sysadmins who are accustomed the unix way of handling – configuring, starting, etc. – applications. We will first introduce a way to make `Python` scripts as `shell` commands and give them arguments optional and mandatory ones.

Deploying The Books Application as a Program

- ▷ **Note:** Having a `Python` script `booksapp.py` you start with `python3 booksapp.py` is sufficient for development.
- ▷ If you want to deploy it on a `web server`, you want more: The sysadmin you deliver your `web application` to wants to start and manage it like any other `UNIX` command.
- ▷ **After all**, your `web server` will most likely be a `UNIX` (e.g. `linux`) computer.
- ▷ In particular behavioural variants should be available via command line options.
- ▷ **Example 10.4.1.** To run the books application without output (`-q` or `--quiet`) and initialized with the seven book records we want to run


```
booksapp -q --initbooks
```

Deploying The Books Application as a Program

- ▷ **Example 10.4.2.** If we forget the options, we need help:

```
> booksapp --help
Usage: <yourscript> [options]

Options:
  -h, --help show this help message and exit
  -q, --quiet don't print status messages to stdout
  -l FILE, --log=FILE write log reports to FILE
  --initbooks initialize with seven book records
```

Deploying a Python Script as a Shell Command/Executable

- ▷ We can make our a `Python` script behave like a native `shell` command.
- ▷ The `file extension` `.py` is only used by convention, we can leave it out and simply

call the file booksapp.

- ▷ Then we can add a special **Python comments** in the first line

```
#!/usr/bin/python3
```

which the **shell** interprets as “call the program python3 on me”.

- ▷ Finally, we make the file hello executable, i.e. tell the **shell** the **file** should behave like a **shell command** by issuing

```
chmod u+x booksapp
```

in the **directory** where the file booksapp is stored.

- ▷ We add the **line**

```
export PATH="./:${PATH}"
```

to the file .bashrc. This tells the **shell** where to look for programs (here the respective **current directory** called .)

Working with Options in Python

- ▷ We have the optparse library for dealing with command line options (**install with pip3**)

- ▷ **Example 10.4.3 (Options in the Books Application).**

```
from optparse import OptionParser
parser = OptionParser()
parser.add_option("-l", "--log", dest="logfile",
                 help="write logs to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                 action="store_false", dest="verbose", default=True,
                 help="don't print status messages to stdout")
parser.add_option('--version', dest="version", default=1.0, type="float",
                 help="the version of the books application")

options, args = parser.parse_args()
# do something with the options and their args.
print ('VERSION: ', options.version)
```


Chapter 11

Image Processing

We will now begin a new topic on our way to a useful image database. In particular we will see how **computer scientists** think about **images**, how **digital images** are represented in **computer memory** and what we can do with them.

11.1 Basics of Image Processing

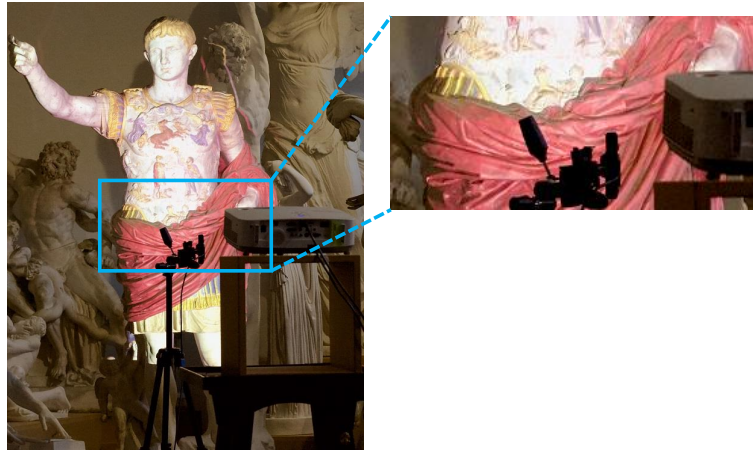
11.1.1 Image Representations

Images

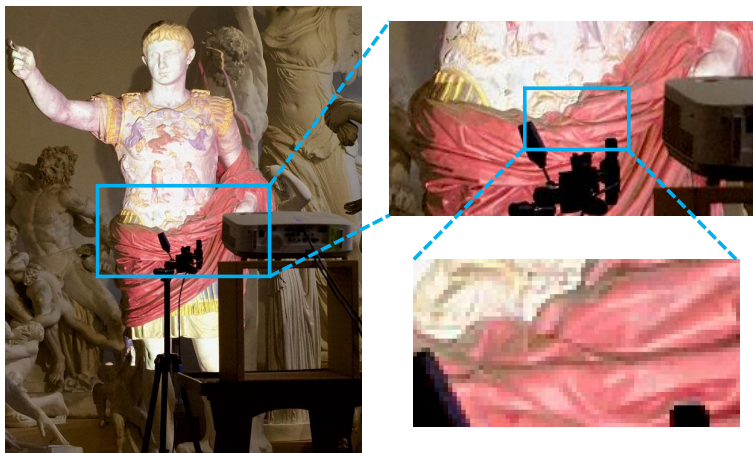
- ▷ **Example 11.1.1 (Zooming in on Augustus).** A **digital image** taken by a standard DSLR camera. Let's zoom in on it!



And a bit more



When zooming in on an **image**, we start to see blocks of colors, which are organized in a regular grid.

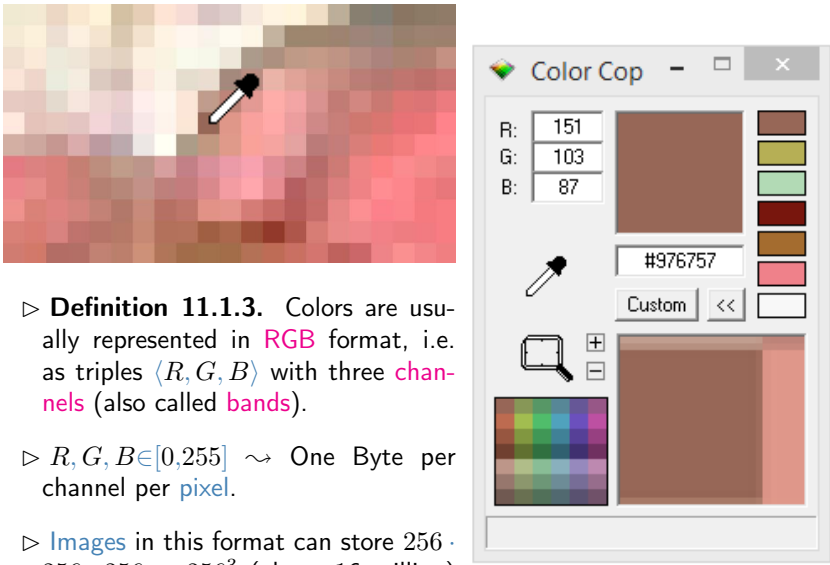


Images as Rasters of Pixels



- ▷ If we zoom in quite a bit more, we see
- ▷ **Observation:** The colors are arranged in a two-dimensional grid (**raster**).
- ▷ **Definition 11.1.2.** We call the grid **raster** and each entry in it **pixel** (from “picture element”).



Colors



- ▷ **Definition 11.1.3.** Colors are usually represented in **RGB** format, i.e. as triples $\langle R, G, B \rangle$ with three **channels** (also called **bands**).
- ▷ $R, G, B \in [0, 255] \rightsquigarrow$ One Byte per channel per **pixel**.
- ▷ **Images** in this format can store $256 \cdot 256 \cdot 256 = 256^3$ (about 16 million) colors.


Michael Kohlhase: Inf. Werkzeuge @ G/SW 2
301
2024-02-08


Each **pixel** stores color information. We can obtain the **values** stored in **digital images** using a color picker. **Image processing programs** like Microsoft Paint or Adobe Photoshop provide color pickers (pipettes), but there also exist standalone applications. In this example we are using Color Cop ¹.

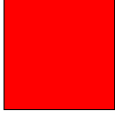
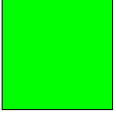
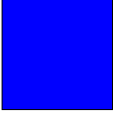
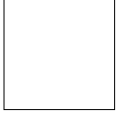
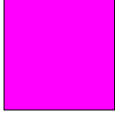
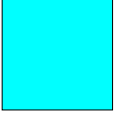
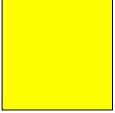

According to the color picker, our **pixel** stores the value (151, 103, 87). Colors are organized in the so-called RGB format, meaning a color is composed from a mixture of red (R), green (G) and blue (B). We call these components **channels** or **bands**.

The value in each of these channels typically ranges from 0 to 255. This is because a single Byte can store exactly this value range and a Byte was deemed enough for most applications. We can deduce that a **pixel** has $256 \times 256 \times 256$ distinct value combinations, which is just over 16 million colors an **image** in this format can display. You might have seen this number on product descriptions of **computer monitors** or cameras.

Color Examples

- ▷ **Example 11.1.4.** A color can be represented by three numbers.

¹<http://colorcop.net/>

			
(255, 0, 0) Red	(0, 255, 0) Green	(0, 0, 255) Blue	(255, 255, 255) White
			
(255, 0, 255) Magenta	(0, 255, 255) Cyan	(255, 255, 0) Yellow	(128, 128, 128) Gray

▷ **Definition 11.1.5.** A color is called **grayscale**, iff $R = G = B$

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 302 2024-02-08

A channel value of 0 means no intensity in this channel, a value of 255 corresponds to full intensity. Thus, in order to create a pure red we set the R channel to 255 and the other two to 0 (no green or blue). Other colors are achieved in a similar fashion.

Secondary colors (e.g. magenta, cyan, yellow) are created by mixtures of red, green, and blue. For example, we create magenta by mixing red and blue.

Different shades of gray are obtained, when $R=G=B$. White is the brightest gray we can achieve, by setting all values to 255. Black on the other hand has all channels set to 0 (meaning no light/intensity).

When processing colors it is often beneficial to think about **normalized colors**. We normalize colors by dividing by 255 (the highest value). Resulting color values are now between 0 and 1.


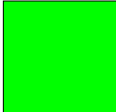



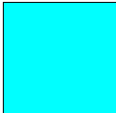
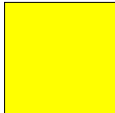

Normalized Color Values

▷ **Observation 11.1.6.** For color representations, only the relative contribution of the *band* is important.

▷ **Definition 11.1.7.** **Normalized colors** use **pixel** values between 0 and 1.

▷ **Idea:** Values are still stored as Bytes, but normalized before use: $v' = v/255$

▷ **Example 11.1.8.**

			
(1, 0, 0) Red	(0, 1, 0) Green	(0, 0, 1) Blue	(1, 1, 1) White
			
(1, 0, 1) Magenta	(0, 1, 1) Cyan	(1, 1, 0) Yellow	(0.5, 0.5, 0.5) Gray

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

303

2024-02-08

HTML Color Codes

- ▷ HTML uses a shorthand notation for colors using hexadecimal numbers.
- ▷ **Example 11.1.9.**

<code>#FF0000</code>	<code>#00FF00</code>	<code>#0000FF</code>	<code>#FFFFFF</code>
Red	Green	Blue	White
<code>#FF00FF</code>	<code>#00FFFF</code>	<code>#FFFF00</code>	<code>#808080</code>
Magenta	Cyan	Yellow	Gray

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

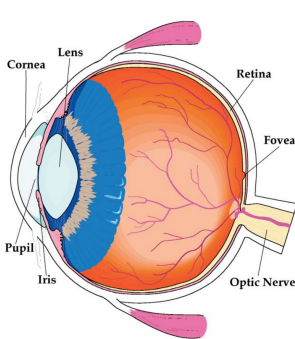
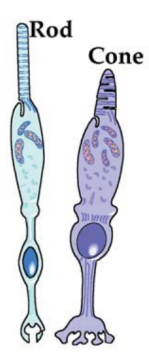
304

2024-02-08

Recall from last semester: In **HTML** and **CSS** we often express colors in **HTML** color codes. This is the same principle as before, however the values are not expressed in decimal numbers but instead in hexadecimal. Quick detour into the real world: Let's explore where the RGB format comes from.

The Human Eye

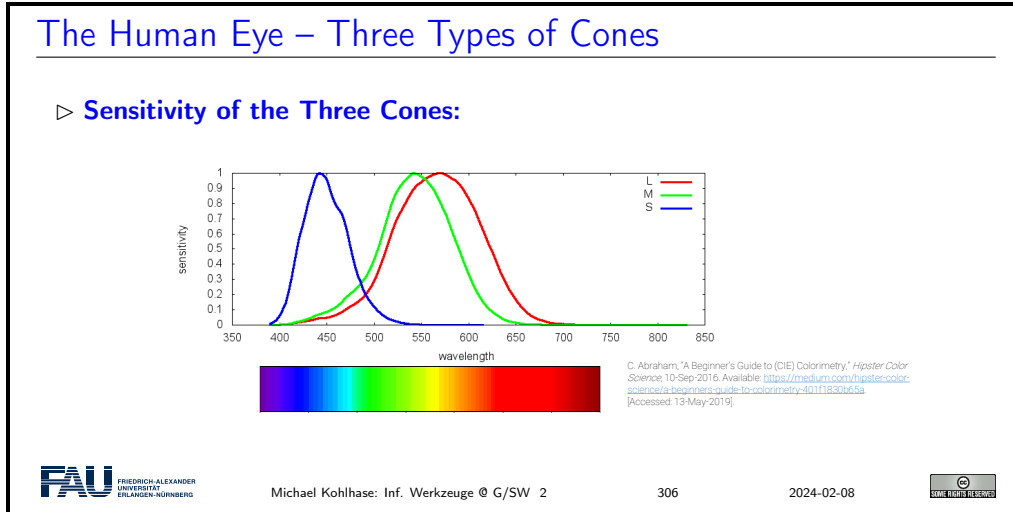
- ▷ **Definition 11.1.10 (The Human Eye).** Light from our surroundings enters our eye through the **lens** and then hits the **retina** on the back of our eye.

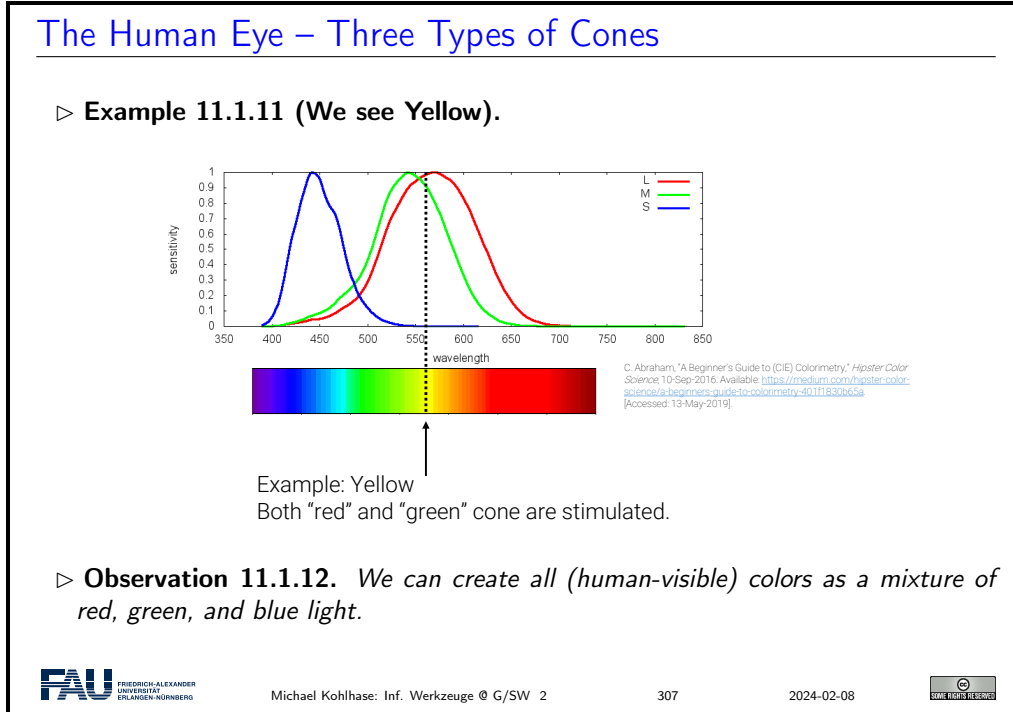
The **retina** has **cones** and **rods**, which are responsible for color and brightness vision, respectively.

- ▷ Since we are interested in colors here, we will ignore the **rods** for the purpose of this lecture.

Light is an electromagnetic radiation. Only a small part of this radiation is visible to the human visual system (wavelengths around 380 to 740 nanometers).



There are three types of **cones**, which react to different areas in this spectrum. They roughly correspond to the wavelengths, which we perceive as red, green, and blue (or rather long, middle, and short wavelengths).



When we now see yellow light for example, the two cones responsible for long and medium length wavelengths are stimulated. Our brain converts this stimulus to yellow.

However, let's imagine we perceive a mixture from red and green light. In this case these two cones will be stimulated, too! Our brain is incapable of distinguishing between these two scenarios, since the physical stimulus on our eye is the exact same!

Monitors take advantage of this, since they usually also have **pixels**.

Monitors

- ▷ **Definition 11.1.13.** A **computer monitor** (or just **monitor**) is an output device for visual information.
- ▷ **Monitors** (usually) have **pixels**, too!
- ▷ **Definition 11.1.14.** In color **monitors**, **pixels** typically consist not of a single light source, but three distinct **subpixels**.
- ▷ If these **subpixels** are small enough and close together, our eye cannot see that the light actually comes from different points and thus perceives the mixture color.

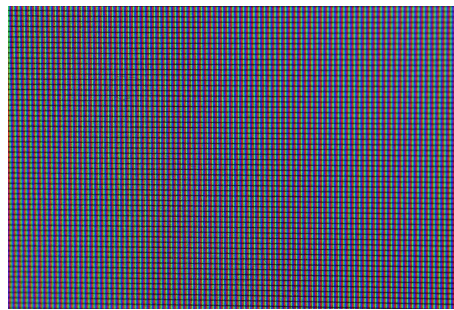
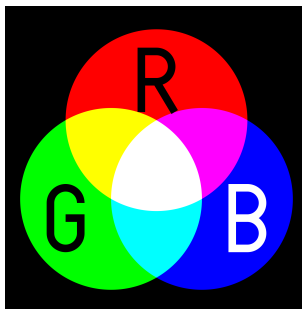
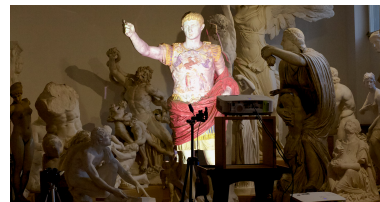


Image Size

- ▷ **Example 11.1.15 (Augustus again).**

Image: 1440×746 pixels
 Expected file size:
 Width · Height · Channels
 $1440 \cdot 746 \cdot 3 = 3,222,720\text{B} \approx 3\text{MiB}$



- ▷ But if we look onto our disk we see something completely different:

Augustus.jpg	4/30/2019 2:58 PM	JPEG image	404 KB
Augustus.png	6/3/2019 12:19 PM	PNG image	1,628 KB

- ▷ On disk, images are usually **compressed** (JPEG, PNG, GIF, WebP etc.). **JPEG file size** is smaller than **PNG**, but **image quality** is lost.

This is because **images** on disc are usually **compressed** and stored in an **image file format** like **JPEG** or **PNG**. Be careful with **JPEG compression**! **JPEG** sacrifices **image quality** in order to achieve smaller **file sizes**!

JPEG Compression Artefacts

- ▷ **Example 11.1.16 (Augustus again).** Here, the Augustus image is saved with a very high jpeg compression. The file size is tiny (27 KB, compare to 440 KB on previous slide). However, the image quality suffers.

JPEG creates blocks of pixels, and approximates the colors in this block with as few bits as possible (according to compression ratio).



 AugustusCompressed.jpg

6/7/2019 9:11 AM

JPEG image

27 KB

In this example we turned the JPEG compression very high, which leads to a tiny file size but strong artefacts in the image quality.

11.1.2 Basic Image Processing in Python

When processing digital images programatically, we have to load them from disc and then perform operations on them. In IWGS we will use Pillow library for this task. The example shows how images are loaded from disc.

The Pillow Library for Image Processing in Python

- ▷ We will use the Pillow library in IWGS.
- ▷ **Definition 11.1.17.** Pillow is a fork (a version) of the old Python library PIL (Python Image Library). (hence the name)
- ▷ Details at <https://pillow.readthedocs.io/slides/stable/>
- ▷ **Install:** `pip install Pillow`
- ▷ **Example 11.1.18.** Determine the color of a particular pixel

```
from PIL import Image
# load image
im = Image.open('image.jpg')
im.show()
```

```
# access color at pixel (x, y)
x = 15
y = 300
r, g, b = im.getpixel((x, y))
```

▷ **Example 11.1.19.** Directly use the `image` object in `jupyter notebooks`:

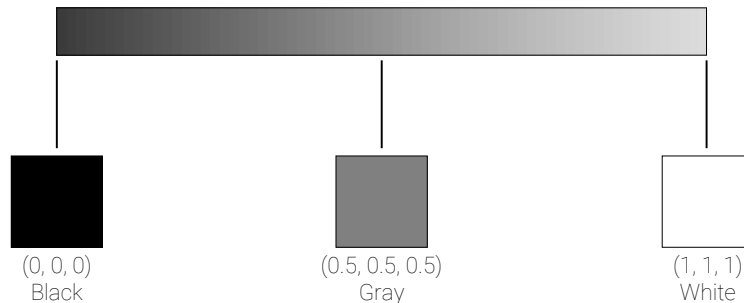
```
from PIL import Image
# load image
im = Image.open('image.jpg')
im # in Jupyter Notebooks, we can directly use the variable
```

The `notebooks` shows the `image` in a new `cell`.

Loading here means that the file is read, and that the `compression` is reversed, i.e. the `digital image` is `decompressed`. This means that the `image` which was before stored in `JPEG compression` is now present in `memory`. You can think about the loaded `image` as a long `Python` list of `pixel values`, i.e. one `pixel` after the other.

Grayscale Images

▷ **Recall:** A color is `grayscale`, iff $R=G=B$.



▷ **Idea:** If all `channels` have the same `value`, why store all three?

▷ `Grayscale images` usually have only one `channel`.

Since it is pointless to store each `value` three times, `grayscale images` usually only store one `value` per `pixel`, which is then tripled before display.

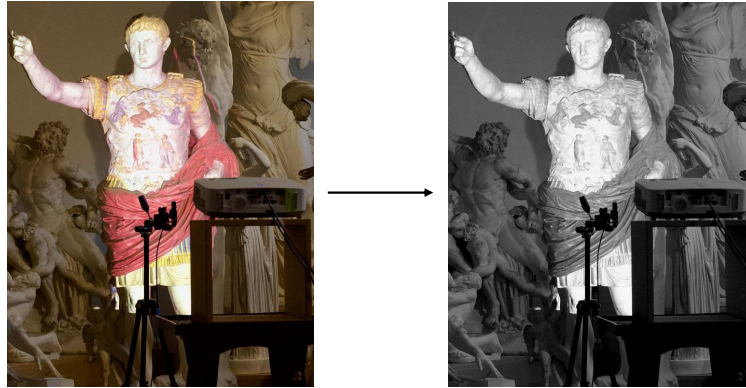
Conversion from color to `grayscale images` is a common operation, which most `image processing` tools (Photoshop etc.) support. It serves as a first example of what we can do with `digital images`.

Grayscale Conversion

▷ **Observation 11.1.20.** *Humans are very sensitive to green, less to red, and least to blue.*

▷ **Definition 11.1.21.** To convert an `image` to an `grayscale image` (`grayscale conversion`), we compute $Gray = 0.21R + 0.71G + 0.08B$

▷ Example 11.1.22 (Grayscale Conversion).



Grayscale conversion is a *weighted sum* of the three channel values. This means, each channel value is multiplied with a *factor* and then the values are *summed* up to form a single value. Since humans are very sensitive to green, the G channel has the highest weight.

We now show some more image operations.

More Image Operations

▷ Example 11.1.23 (More Image Operations).



Original



Grayscale



Sepia



Inverse

Each pixel is
processed separately!



Threshold



Red Channel
Extraction

▷ As for grayscale conversion of these process each pixel separately.

Implementation of these operations is very simple in Python. Since we store all our pixels in a large list in Pillow, we can simply create a for-loop over this list, do our calculation and store the result in a new image at the same pixel coordinate.

Image Operations in Pillow

▷ The [pillow library](#) supports many [image operations](#) out of the box.

▷ **Example 11.1.24 (Grayscale Conversion and Inversion in Pillow).**

```
from PIL import Image, ImageOps
im = Image.open('image.jpg')
# convert to grayscale
gray = ImageOps.grayscale(im)
# invert image
inverse = ImageOps.invert(im)
```

▷ Complete List: <https://pillow.readthedocs.io/en/stable/reference/ImageOps.html>

Transparency is an important operation. In this example we want to layer two [digital images](#) on top of each other. We thus need to store for each [pixel](#) a measure of how transparent it is.

We expand our RGB notion to RGBA, by introducing a fourth channel A. A stands for alpha and corresponds to the [opacity](#) of a [pixel](#), i.e. a value of 0 means zero [opacity](#) (fully [transparent](#)), a value of 1 (normalized) means fully [opaque](#) (no [transparency](#)).

Transparency and Image Composition

▷ Sometimes we want to overlay [images](#) \rightsquigarrow [layers](#).

▷ We need a notion of how transparent a [pixel](#) is.

▷ **Definition 11.1.25.** We introduce a fourth [channel](#): A (for [alpha](#)). Alpha is the [opacity](#) (inverse of [transparency](#)). A [pixel](#) is now $\langle R, G, B, A \rangle$.

▷ **Example 11.1.26 (Combining Images).**



▷ **Note:** The order of layers is important here: The Augustus [image](#) is below the other [image](#)! The Augustus [image](#) has *no* transparency, the second [image](#) does!

See examples for the [opacity](#) here. Fully transparent regions (visualized by the checkerboard), have an alpha value of 0. Fully opaque regions have a value of 1. Intermediate values are possible which correspond to partial transparency.


Transparency (continued)

▷ **Example 11.1.27 (Combining Images).**

(R,G,B,A) = (0, 0, 0, 0)
Full transparent

(R,G,B,A) = (0.6, 0.0, 1.0, 0.5)
Half transparent purple

(R,G,B,A) = (1, 1, 0, 1)
Full yellow



$$R_{\text{target}} = (1-A) \times R_{\text{augustus}} + A \times R_{\text{purple,yellow}}$$

$$G_{\text{target}} = (1-A) \times G_{\text{augustus}} + A \times G_{\text{purple,yellow}}$$

$$B_{\text{target}} = (1-A) \times B_{\text{augustus}} + A \times B_{\text{purple,yellow}}$$

FAU
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

317

2024-02-08

CC BY-NC-SA

The final *image* is then composed by deciding for each *pixel* how much color from each source *image* should contribute. Note that this is again a *per-pixel* operation, which can easily be *implemented* with a simple for-loop.

11.1.3 Edge Detection

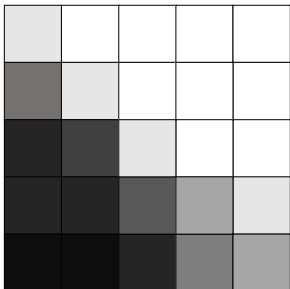
We will now look at more interesting *image operations*. A typical example especially important for object recognition in *digital images* is to find *features* i.e. areas in the *image*, which are recognizable.

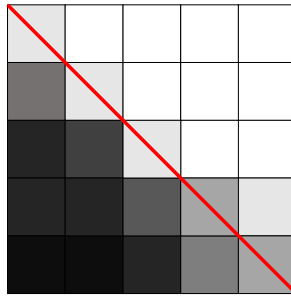
For example, let's say we want to find so-called *edges* in our *image*, i.e. areas where the color changes rapidly. *Edges* often correspond to object outlines. We will see an example later.

Edge Detection

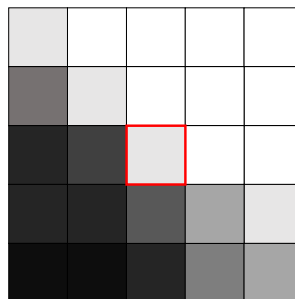
▷ **Goal:** Find interesting parts of *image* (*features*).

▷ **Example 11.1.28 (Edge Detection).** Find *edges*, i.e. *image* sections, where color changes rapidly.

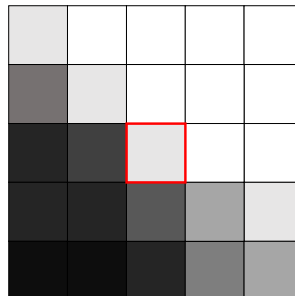




Clearly there is an edge in this **image**. How do we detect it automatically?



Decide for each **pixel**, whether it is on an edge. Here: Is marked **pixel** an edge **pixel**?



Inspect neighbor **pixels**.

▷ **Definition 11.1.29.** We call a **pixel** a **horizontal edge pixel**, iff

$$I_B - I_T + I_{BL} - I_{TL} + I_{BR} - I_{TR} > \tau$$

for some threshold τ and a **vertical edge pixel**, iff

$$I_R - I_L + I_{TR} - I_{TL} + I_{BR} - I_{BL} > \tau$$

In this (admittedly simple) example **image**, we can clearly see, that there is an edge present, where the color shifts fast from dark to light. We will now explore, how we can detect such an edge automatically.

The idea is to decide for each **pixel** if it is part of an edge or not (binary decision, yes or no). Let's take the marked **pixel** as example, but remember that the following operations are performed on each **pixel** in the **image**.

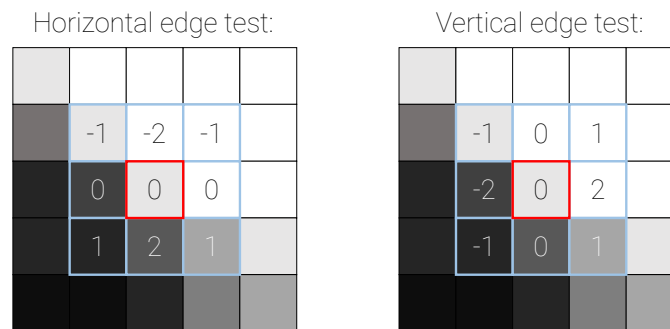
The idea for this edge detection algorithm is to compare the pixel column left to our marked pixel to the column to the right. If the difference between the two columns is large, we know that we are observing a vertical edge.

Analogous we can do the same for horizontal edges, by comparing the row above to the row below our marked pixel.

We could perform this operation using only the pixels marked by L, R, B, and T, so only the direct neighbors. By taking the diagonal pixels into consideration, too, we make sure we only detect larger features.

Algorithm: Sobel Filter

- ▷ **Idea:** There is a general algorithm that computes this.
- ▷ **Definition 11.1.30.** Given a 3×3 matrix M , the Sobel filter computes a new pixel value by getting the pixel value of each neighbor in 3×3 window, multiply with the components in M and adding everything up.
- ▷ **Observation 11.1.31.** Given a suitable matrix M , the Sobel filter computes the quantities from Definition 11.1.29.
- ▷ **Example 11.1.32 (Edge Tests via Sobel Filters).**



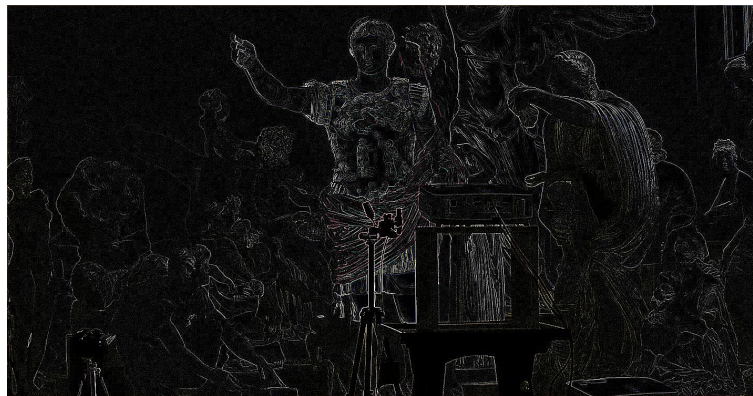
The operation we described here is called Sobel filter, named after Irwin Sobel.

Usually the direct neighbors are deemed more important than the diagonal neighbors. The pixel values of the neighbor pixels are thus weighted, such that the direct neighbors contribute more.

Here we see an example of edge detection. White pixels in the right image are pixels, which were classified as edge pixels, i.e. pixels where large changes in color are present. Black pixels are no edges.

Edge-Detection in Pillow

- ▷ **Example 11.1.33 (Augustus and his Edges).**



▷ **Example 11.1.34 (Edge Detection in Pillow).**

```
from PIL import Image, ImageFilter
im = Image.open('augustus.jpg')
edges = im.filter(ImageFilter.FIND_EDGES)
edges.show() # or just edges in Jupyter
```

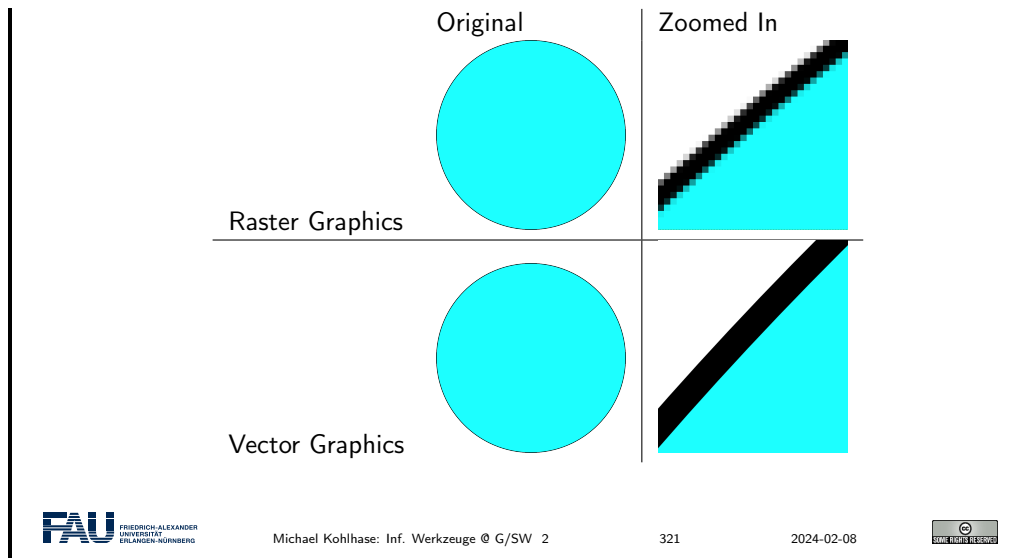
11.1.4 Scalable Vector Graphics

The **digital images** we talked about so far store colors in a large grid of **pixels** (a raster). A common problem with these types of **images** is that we cannot zoom in on them as far as we want, without losing quality. At a certain point we start to see the individual **pixels**.

Vector graphics are an alternative way of storing **digital images**, which solve this problem.

Vector Graphics

- ▷ **Problem:** **Raster images** store colors in **pixel** grid. Quality deteriorates when **image** is zoomed into.
- ▷ Vector Graphics solve this problem!



The idea of vector graphics is fundamentally different than the idea of raster graphics. Instead of storing **pixels**, we now store shape information!

For example, for a circle we don't store a color for each **pixel**, but we rather just store where the circle is, along with its radius, color, etc.

Vector Graphics (Definition)

- ▷ **Definition 11.1.35.** Image representation formats that store shape information instead of individual **pixels**, are referred to as **vector graphics**.
- ▷ **Example 11.1.36.** For a circle, just store
 - ▷ center
 - ▷ radius
 - ▷ line width
 - ▷ line color
 - ▷ fill color
- ▷ **Example 11.1.37.** For a line, store
 - ▷ start and end point
 - ▷ line width
 - ▷ line color

Note that most monitors cannot display vector graphics. There are vector monitors, but they are not common.

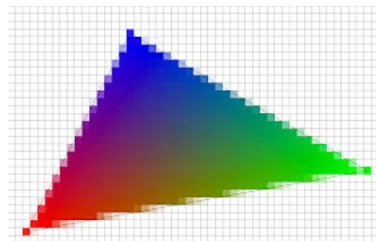
Vector Graphics Display

- ▷ There are devices that directly display vector graphics.
- ▷ **Example 11.1.38.**



▷ **Definition 11.1.39.** For monitors, vector graphics must be rasterized – i.e. converted into a raster image before display.

▷ **Example 11.1.40.**



The monitor displayed in Example 11.1.38 here does not have pixels. It instead moves a laser and traces a polygon (the asteroids and spaceship). The laser stimulates a phosphor layer, which then glows.

Common monitors work with pixels. Vector graphics are thus rasterized (i.e. turned into raster graphics) just before being displayed. The rasterizer decides for each pixel, whether it is inside or outside the shape and thus what RGB value to display.

On the edges of Example 11.1.40, we see pixels whose barycenter is outside the triangle but that are colored in a very light variant of the adjoining pixels. This technique is called anti-aliasing and is used to make the jagged lines created by rasterization less noticeable to the human eye.

We now introduce a concrete representation format for vector graphics.

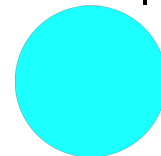
SVG is one image format for vector graphics. Since it is XML based we are able to read it. As described above, we can create circles by specifying a position, radius, and style (color etc).

Scalable Vector Graphics (SVG)

▷ **Definition 11.1.41.** Scalable Vector Graphics (SVG) is an XML-based markup format for vector graphics.

▷ **Example 11.1.42.**

```
<svg xmlns="http://www.w3.org/2000/svg"
  width="100" height="100" >
  <circle cx="50" cy="50" r="50"
    style="fill:#1cffff;stroke:#000000;stroke-width:0.1" />
</svg>
```



- ▷ The <svg> tag starts the **SVG** document, width, height declare its size.
- ▷ The <circle> tag starts a circle. cx, cy is the center point, r is the radius. style describes how the circle looks.

As the **SVG** size is 100x100 and the circle is at (50,50) with radius 50, it is centered and fills the whole region.

More SVG Primitives

- ▷ **Example 11.1.43 (Rectangle).**

```
<rect x="..." y="..." width="..." height="..." style="..." />
```

- ▷ **Example 11.1.44 (Ellipse).**

```
<ellipse cx="..." cy="..." rx="..." ry="..." style="..." />
```

- ▷ **Example 11.1.45 (Line).**

```
<line x1="..." y1="..." x2="..." y2="..." style="..." />
```

- ▷ **Example 11.1.46 (Text).**

```
<text x="..." y="..." style="...">This is my text!</text>
```

- ▷ **Example 11.1.47 (Image).**

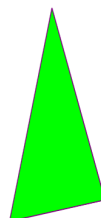
```
<image xlink:href="..." x="..." y="..." width="..." height="..." />
```

We can draw arbitrary polygons by specifying a list of coordinates.

SVG Polygons

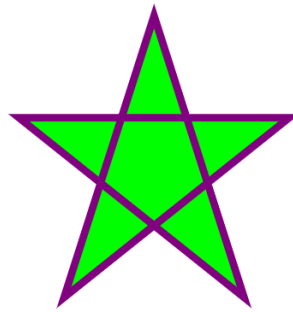
- ▷ **Example 11.1.48 (An SVG Triangle).**

```
<svg height="210" width="500" xmlns="http://www.w3.org/2000/svg">
  <polygon points="200,10 250,190 160,210"
    style="fill:lime;stroke:purple;stroke-width:1"/>
</svg>
```



▷ **Example 11.1.49 (An SVG Pentagram).**

```
<svg height="210" width="210" xmlns="http://www.w3.org/2000/svg">
  <polygon points="100,10 40,198 190,78 10,78 160,198"
    style="fill:lime;stroke:purple;stroke-width:5;fill-rule:nonzero;" />
</svg>
```



SVG can directly be embedded in [HTML](#)!

SVG in HTML

- ▷ SVG can be used in dedicated files (file ending `.svg`) and referenced in a `` tag.
- ▷ It can however also be written directly in [HTML](#) files.
- ▷ **Example 11.1.50.** Triangle from Example 11.1.48 embedded in [HTML](#) file

```
<html>
  <body>
    <svg height="210" width="500" xmlns="http://www.w3.org/2000/svg">
      <polygon points="200,10,250,190,160,210"
        style="fill:lime;stroke:purple;stroke-width:1" />
    </svg>
  </body>
</html>
```

We now explore a useful attribute of [SVG](#) called `viewBox`. We said that we can zoom in onto vector graphics as far as we want without losing quality, so let's give ourselves the possibility to do so.

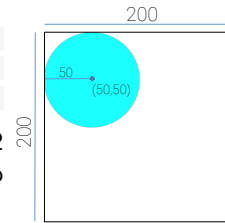
The SVG `viewBox` Attribute

- ▷ **Idea:** The [SVG](#) `viewBox` attribute allows us to zoom into an [image](#).

▷ **Example 11.1.51.**

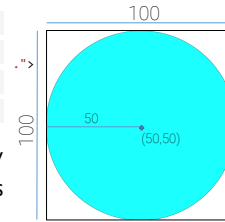
```
<svg width="200" height="200" xmlns="...">
  <circle cx="50" cy="50" r="50" style="..." />
</svg>
```

Here, the width and height are scaled by a **factor** of 2 to give us a little more room. Sometimes we want to specify a larger **image**, but only display a section of it.

▷ **Example 11.1.52.**

```
<svg width="200" height="200" xmlns="..."
  viewBox="0,0,100,100" >
  <circle cx="50" cy="50" r="50" style="..." />
</svg>
```

`viewBox` specifies a region inside our canvas. Only things inside that are drawn. The resulting **image** is then stretched to the canvas size (zoom effect).



The top example shows a 200 by 200 units large **SVG** canvas. In the top left quadrant we draw a circle.

The second code snippet employs the `viewBox` attribute, which specifies an area of the **image** we want to display. In this example we give it a region from (0,0) to (100,100), meaning we specify exactly this upper left quadrant.

`viewBox` now does two things: First, it only draws objects inside this region, i.e. it discards everything outside. Second, it stretches this region to the whole **SVG** canvas. This means, that our final **image** is still 200 by 200 units (**pixels**) in size, but we only see a region of our original **image**. This gives a zoom effect.

11.2 Project: An Image Annotation Tool

Project: Kirmes Image Annotation Tool

- ▷ **Problem:** Our Books-App project was a fully functional **web application**, but does not do anything useful for DigiHumS.
- ▷ **Idea:** Extend/Adapt it to a database for **image annotation** like LabelMe [LM].
- ▷ **Setting:** Prof. Peter Bell (formerly at FAU) conducts research on baroque paintings on parish fairs (Kirmes) and the iconography in these paintings. We want to build an annotation system for this research.
- ▷ **Project Goals:**
 1. Collect kirmes **images** in a **database** and display them,
 2. mark interesting areas and provide meta data,
 3. display/edit/search annotated information.

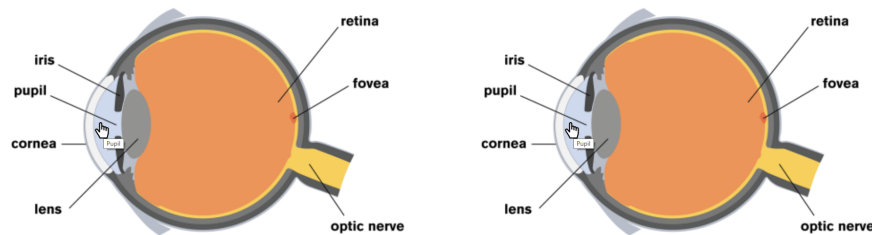
1. is analogous to Books-App, for 2/3. we need to know more
- ▷ **Plan:** Lern the necessary technologies in class, build the system in exercises

In our quest for an [image annotation](#) technology, we will first explore [HTML image maps](#).

HTML Image Maps

▷ **Definition 11.2.1.** [HTML image maps](#) mark [areas](#) in an [digital image](#) and assign names and links to them.

▷ **Example 11.2.2.** An [image map](#) adds hover and on click behavior



Clicking on the pupil leads to:

<https://en.wikipedia.org/wiki/Pupil>

Clicking on the vitreous body leads to:

https://en.wikipedia.org/wiki/Vitreous_body

```
<html>
  <body>
    
    <map name="image-map">
      <area title="Pupil"
        href="https://en.wikipedia.org/wiki/Pupil"
        coords="102,117,143,219" shape="rect" />
      <area title="Vitreous_Body"
        href="https://en.wikipedia.org/wiki/Vitreous_body"
        coords="242,166,107" shape="circle" />
    </map>
  </body>
</html>
```

▷ Easy creation of [image maps](https://www.image-map.net/): <https://www.image-map.net/>

[Image maps](#) provide a way to mark areas in an [image](#). These areas act as links, i.e. clicking on them leads to different [URLs](#). For example in this case there are two regions in the [image](#) (pupil and vitreous body), which - when clicked on - direct your browser to the respective Wikipedia articles.

`` tag specifies [image](#) as always, but we no add a new attribute `usemap` that specifies the name of an [image map](#) to use (here `image-map`).

The map itself is defined by the `<map>` element (with the same name!). Inside the map we define our areas for the two parts of the eye we want to annotate. In this example we use a rectangle for the pupil and a circle for the vitreous body.

This is specified by the two `<area>` elements, which have a `title` attribute (shown on hover) and a link (`href`). The shapes are specified by the `shape` attribute with values `rect`, `circle`, `poly`, ... and some coordinates specified in the `coords` attribute.

[Image maps](#) are useful for certain tasks, but aren't quite what we want for our annotation tool. They are somewhat difficult to work with, especially if you want the areas to react to your mouse.

Problems of HTML Image Maps

- ▷ **Problem:** Image maps do not allow interaction:
 - ▷ the name attribute can only contain unstructured information.
 - ▷ no integrated highlight for image maps area,
 - ▷ no onclick or onmouseover attributes.
- But the whole point is to have (arbitrarily) complex metadata for image regions.
- ▷ **New Plan:** Use a newer technology: SVG and CSS.

We therefore go a different route, by using SVG and CSS: The whole functionality of the annotation tool will be implemented in a single SVG image where CSS provides the interactivity. First we implement the equivalent of an image map by including a raster graphic (our image) and four rectangles for the annotation areas. Coordinates of the rectangles can be read out from any tool like Microsoft Paint or GIMP.

Handcrafting better Image Annotations with SVG and CSS

- ▷ **Idea:** Integrate the image and the areas into one SVG and make areas interactive via CSS.
- ▷ **Example 11.2.3 (Paper Prototype).** Highlight regions and display information on hover.



George Washington



Abraham Lincoln

Displayed here is our goal behavior, which we will pursue on the following slides. As we have not implemented this, we could have created this in an image processing program, e.g. photoshop or GIMP. We call such a mockup for informing our design intuition a paper prototype.

The rectangles mark certain parts of our image and react to the mouse being moved over them. On the one hand the area is highlighted by the white rectangles. Additionally descriptive text is displayed below the image (in this case the name of the respective president).

SVG Annotation Implementation Areas

- ▷ **Implementing Areas as Rectangles:**

```

<svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024" >
  <!-- Image -->
  <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />
  <!-- Areas in image as rects. -->
  <rect x="300" y="125" width="250" height="300"/>
  <rect x="550" y="225" width="200" height="300"/>
  <rect x="750" y="375" width="200" height="300"/>
  <rect x="999" y="375" width="200" height="300"/>
</svg>

```

Add four <rect>s (one for each president).

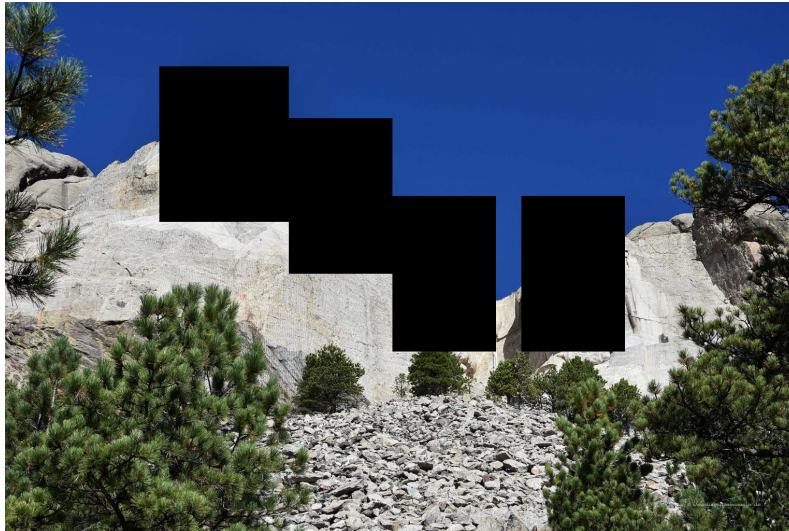
Note again: The [image](#) is **not** a [vector graphic](#). Even though it is embedded in a [SVG](#) environment, it will not have the benefits of a [vector graphic](#), i.e. it will lose quality when zoomed in on.

Note furthermore: the order of elements in our [SVG](#) matters! Here the <rect> tags are specified *after* the [image](#). [SVG](#) draws the elements from top to bottom. The rectangles are therefore drawn on top of the [image](#).

Swapping this order would lead to the [image](#) being drawn on top of the rectangles. This means, that the rectangles would not be visible!

SVG Annotation Implementation Result

▷ **Areas as Rectangles – Result:** Now the rectangles are visible



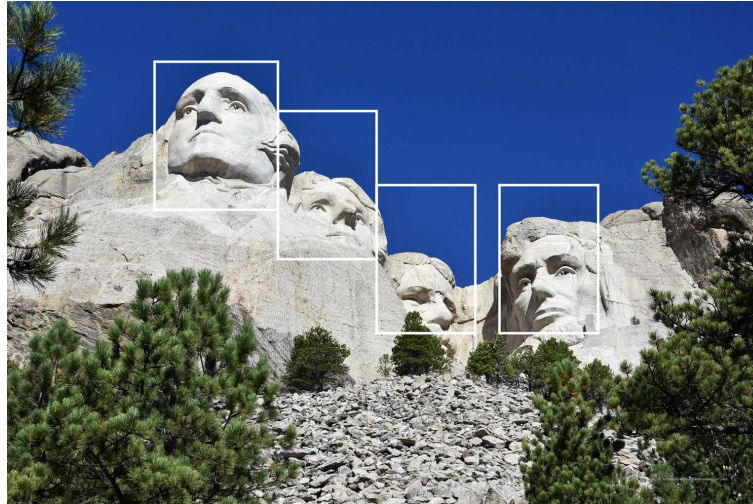
The rectangles are now visible in our [SVG](#). Their color defaults to black, so let's fix this next, so that we can actually see our [image](#) again.

We add a [CSS](#) stylesheet to our site. This can either be defined in a separate file (like in this example), or be specified directly in the [HTML](#) inside of <style> tags.

Adding CSS for the Areas

▷ **Example 11.2.4 (Adding CSS).**

```
rect {fill-opacity:0; stroke:white; stroke-opacity:1; stroke-width:5px}
```



Our goal is to give the rectangles a solid white border, but no inner color. We thus change the stroke (border) parameters.

The fill opacity is set to zero, in order to make it completely transparent so we see the presidents' heads again. However, the rectangles are always visible and do not react to our mouse input. We will fix this next.

Selectively Highlighting Areas

- ▷ **Problem:** Now the rectangles are always visible.
- ▷ **Idea:** make the rectangles invisible by default only show them on hover.
- ▷ **CSS:** We set the stroke **opacity** to zero by default and add a hover **selector**.

```
rect {fill-opacity:0; stroke:white; stroke-opacity:0; stroke-width:5px}  
rect:hover {stroke-opacity:1}
```



The [hover selector](#) of the rectangles specifies their style, whenever the mouse is over the element. This allows us to specialize the appearance for this case: we set the opacity back to one, meaning full [opacity](#) and thus visibility.

Net Effect: The rectangles are now invisible, except when hovered over by the mouse.

We will now add the description text to each of our annotation areas. Since our text should appear below the [image](#), let's start by giving ourselves a bit more room in the [SVG](#) canvas. We thus increase the [SVG](#) height by a bit. Note, that this does not impact the [image](#) (because it has an own height).

Adding Annotation Text

▷ **Adding Annotation Text** and making space for it.

```
<svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1224" >
  <!-- Image -->
  <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />
  <!-- Areas in image as rects, text below -->
  <rect x="300" y="125" width="250" height="300" />
  <text x="100" y="1200">George Washington</text>
  <rect x="550" y="225" width="200" height="300" />
  <text x="100" y="1200">Thomas Jefferson</text>
  <rect x="750" y="375" width="200" height="300" />
  <text x="100" y="1200">Theodore Roosevelt</text>
  <rect x="999" y="375" width="200" height="300" />
  <text x="100" y="1200">Abraham Lincoln</text>
</svg>
```

and we add some [CSS](#):

```
text {fill:black; opacity:1; font-size:100px}
```

We then add the text. Note, that all text elements have the exact same position below the [image](#). They only differ in the text they display (the name of the president).

We write each text element directly below the corresponding rectangle tag, for reasons we will explain in a bit!

We also style the text: The text color is specified by the fill attribute. This is the default, so it's not really necessary to specify this. However, oftentimes it is advisable to be as verbose as possible with certain attributes, because this more clearly shows our intention.

Adding Annotation Text – Result

▷ **Adding Annotation Text – Result:**



The text is still unreadable, mainly because all texts are right above each other, but this is expected so far, since we specified all text tags to have the same position. Our main problem is, that the text does not react to our mouse input yet. Remember: Our goal is that each text element is only displayed, when the corresponding rectangle in the [image](#) is hovered by the mouse. Our approach is analogous to the hovering of the rectangles we did previously. We text a default opacity of zero, and a hover opacity of one.

Remember though, that the hover selector always influences the element it is specified on, i.e. when writing `text:hover`, and then changing the opacity, this changes the opacity when we hover over the text, *not* when we hover the rectangle. We thus introduce the [CSS sibling operator](#), `+`.

Selectively Showing Annotations

- ▷ **Problem:** Now the annotations are always visible.
- ▷ **Idea:** Add [CSS](#) hover effect for `<rect>`s, which effects the `|<text>|`.
- ▷ **Definition 11.2.5.** The [CSS sibling operator](#) `+` modifies a selector so that it (only) affects following sibling elements (same level).
- ▷ **Example 11.2.6.** In the [CSS](#) directive

```
rect:hover + text {<rules>}
  ↑           ↑           ↑
  Selector  Sibling operator Target
```

the rules affect the [SVG](#) `<text>` directly after the `<rect>` element.

- ▷ **Again:** the order of elements in the [HTML](#) is important!
- ▷ **CSS:** We set the [opacity](#) to zero by default and add a hover [selector](#) for the following `<text>` sibling.

```
text {fill:black; opacity:0; font-size:100px}
rect:hover + text {opacity: 1}
```

The sibling operator influences the next element of the specified type (in our case text) in the [HTML/SVG](#). This is why earlier we put the text elements always directly after the rectangle.

This way, when a rectangle is hovered over, the next text element is always the corresponding description and will thus become visible.

Image Annotation Tool – Final Result

- ▷ Now our annotation tool works as expected!
- ▷ **Example 11.2.7 (Final Result).** Highlight regions and display information on hover.



George Washington



Abraham Lincoln

11.3 Fun with Image Operations: CSS Filters

Let's explore more the capabilities [CSS](#) has to offer for applying operations to [digital images](#). In this example we make an [image](#) gray, by specifying a [grayscale](#) filter attribute. The argument of the filter gives us the possibility to make the [image](#) only a little gray. Since it is set to 100% in this example, the [image](#) is converted to perfect [grayscale](#).

CSS Image Filters

- ▷ **Goal:** Apply [image filters](#) ([grayscale](#) etc.) directly in [CSS](#).
- ▷ **Example 11.3.1 (Image Effects via inline CSS).**

```

```



- ▷ **Disadvantage:** The original `image` is delivered to client. When user saves the `image`, they get the original!

One extremely important thing to keep in mind is that `CSS` is executed on the client (the user's browser). The original `image` or text is delivered to the client, where the filter is applied. You can try this out by right-clicking a filtered `image` on a `web site` and saving it to your hard drive. Note, that the original `digital image` is saved!

The implication here is, that for certain content it is best to perform the filter on the server and then deliver the filtered content to the user, so that he or she does not even have the possibility to get the original. This however also means more computation on the server, which might be expensive.

Rule of thumb: Perform as much as possible on the `client` (`CSS` and `JavaScript`) and as much as necessary on the `server` (for example `python` in `bottle`).

Here are more examples of `image filters`. The `CSS selectors` here start with dots. This makes them influence `HTML` elements of the respective class name, i.e. the `selector` `.shadow` gives the `HTML` element with class `shadow` a drop shadow.

Some more CSS Filters

- ▷ **Example 11.3.2 (Image Effects via CSS Style sheets).**

```

```



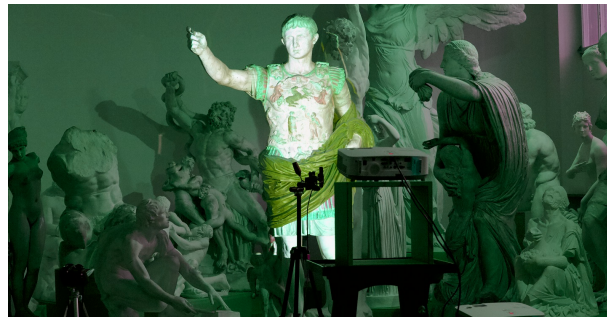
```

```




```

```



Blurring: Blur is an [image operation](#), which mixes each pixel's color with the colors of its neighbor. The operation is thus similar to our edge detection example from earlier, but with different weights per neighbor pixel.

Also, for blur it is possible to specify larger neighborhoods. In this case the radius of our neighborhood is 4 pixels, meaning that we mix the colors of a region with radius 4.

Contrast: Contrast makes dark colors darker and light colors lighter for arguments over 100%. This increases the range between the darkest and lightest pixel.

For arguments under 100%, the contrast shrinks.

Hue Rotation: The color wheel at the top might look familiar to you. It is a standard way of displaying colors. The outer ring is roughly equivalent with the colors of the rainbow (with some exceptions; purple for example is not a rainbow color).

The hue-rotate filter rotates this color wheel, such that each color lands in a different spot. In our example (90deg), red becomes green. This effect can be observed on Augustus' cloak.

Another useful thing is the combination of [CSS filters](#). For example you can blur an [image](#) and then convert it to [grayscale](#), as showcased in the example.

Combining CSS Filters

▷ **Idea:** We can also combine [image filters](#) flexibly. The easiest way is when we define [CSS classes](#) for that.

▷ **Example 11.3.3 (Tie CSS Filters to Classes).**

```
<html>
<head>
  <style type="text/css">
```

```

.blur { filter: blur(4px); }
.brightness { filter: brightness(0.30); }
.contrast { filter: contrast(180%); }
.grayscale { filter: grayscale(100%); }
.huerotate { filter: hue-rotate(180deg); }
.invert { filter: invert(100%); }
.opacity { filter: opacity(50%); }
.saturate { filter: saturate(7); }
.sepia { filter: sepia(100%); }
.shadow { filter: drop-shadow(8px 8px 10px green); }
</style>
</head>
<body>
  
</body>
</html>

```

- ▷ **Note:** The order is important: Changing the order of filters yields different results.

Digital image are not the only **HTML** element which can be filtered. It turns out that you can apply filters to nearly everything in **HTML**, for example text. Note that here we are using the `blur` class from earlier.

Filtering Everything Else

- ▷ **Note:** **CSS** filters don't just apply to **images**! (Almost) everything can be filtered.

- ▷ **Example 11.3.4 (Filtering Text (Blurring)).**

```
<p style="filter: blur(3px)">A severely blurred Text</p>
```

A severely blurred Text

A fun thing to play around with are **CSS** animations.

CSS Animations

- ▷ **Definition 11.3.5.** **CSS animations** change state of an object over time.

- ▷ **Example 11.3.6 (Inverting an image).**

```

img {animation: invertAnimation 1s forwards}
@keyframes invertAnimation {
  from {filter: none}
  to {filter: invert(100%)}
}

```

In this case we define an animation called *invertAnimation* which applies an inversion-filter. The syntax specifies that at the beginning of the animation, no filter should be applied and in the end we want the [image](#) to be completely inverted.

We then apply the animation to all elements of tag ``. We declare that the animation should run one second (1s), so the [image](#) is inverted after one second.

The last attribute specifies what should happen after the animation is completed. `forwards` means that the element should simply stay how it is, so it stays inverted after the one second.

SVG Filters

- ▷ **Note:** Unfortunately in SVG the filtering works differently from CSS.
- ▷ **Example 11.3.7 (Blurring Mt. Rushmore in SVG).**

```



<svg xmlns="http://www.w3.org/2000/svg" width="1536" height="1024">
  <style> image {filter: url(#myCustomFilter)}</style>
  <image width="1536" height="1024" xlink:href="mount_rushmore.jpg" />
  <!-- Image filter -->
  <filter id="myCustomFilter">
    <feGaussianBlur stdDeviation="5" />
  </filter>
</svg>

```
- ▷ **Example 11.3.8 (SVG Filters can be combined).**

```

<filter id="myCustomFilter">
  <feGaussianBlur stdDeviation="5" />
  <feColorMatrix type="saturate" values="0.1" />
</filter>

```


Michael Kohlhase: Inf. Werkzeuge @ G/SW 2
346
2024-02-08


In the first example we define a filter at the bottom. We give it a name (*myCustomFilter*), which we can then reference in the [CSS](#) snippet above. With the `url` function we can apply a filter with the given name to all [images](#).

The *Gaussian Blur* filter here is similar to the *blur* filter in [CSS](#).

Similarly to [HTML](#), we can combine filters in SVG as well. In the second example we apply a saturation filter after the blur. This is similar to a [grayscale](#) filter.

11.4 Exercises

Problem 4.1 (Basic Image Manipulation)

In this exercise we will explore Pillow's [image processing](#) capabilities. Create a new [Python](#) file `ImageManip.py` and import the `Image` and `ImageOps` modules like this:

```
from PIL import Image, ImageOps
```

Write a [Python function](#) `transformImage`, which takes as [arguments](#) a [digital image](#) and a [string](#). The [string](#) describes, which transformation should be applied to the [image](#). For example, if the value of the passed [string](#) is `"gray"`, your function should convert the [image](#) to [grayscale](#) and return the resulting image.

You find a complete list of Pillow's [image processing](#) functions here: <https://pillow.readthedocs.io/mod/stable/reference/ImageOps.html>. Your function should at least support five of them.

You can freely choose the string value you want to assign each operation. For example, if you want to support the grayscale operation, you can choose whether the expected string is supposed to be "gray" or "grayscale" or something else, as long as it is sensible.

If the passed string does not match any operation, just return the original `image`.

Outside the function, load an `image` from your hard drive using Pillow's `Image.open` function. You may use one of the `images` in the Kirmes repository or use one of your own `digital images`.

Test your `transformImage` function by passing the `image`, along with some strings specifying the image operation. Display the transformed `image` using Pillow's `show` functionality.

Refer to the course notes for examples of the `open` and `show` methods.

Problem 4.2 (Watermarking Images)

In this exercise we will add functionality to apply a watermark to a `digital image`. We provide a watermark image (`Watermark.png`) together with this assignment (StudOn and Kirmes repository), but feel free to create one yourself.

Create a new `Python function` `applyWatermarkToImage`, which takes an `image` as argument. In the function, load the watermark image from your hard drive. Then use Pillow's `alpha_composite` function to overlay the watermark on top of the input `image`: https://pillow.readthedocs.io/mod/stable/reference/Image.html#PIL.Image.Image.alpha_composite

Note that there are two versions of `alpha_composite` in Pillow. The one we are using here directly modifies the original `image` and does not return a new one.

At the end of your function, convert the watermarked `image` back to RGB (analogous to above) and return the result.

Test your function and show the watermarked image! You can also use the `save` function to write the `image` to your hard drive:

```
im.save("filename.jpg", "JPEG")
```

Optional for the highly motivated: Check out the following tutorial, if you want to write arbitrary text as watermark: <https://pillow.readthedocs.io/mod/stable/reference/ImageDraw.html#example-draw-partial-opacity-text> Note: When they load a font (`fnt = ImageFont.truetype(...)`), just pass "arial.ttf" as argument (or another font which is `installed` on your PC).

Problem 4.3 (Putting Thumbnails in Database)

Our image database and front-end are taking shape. On the home page we currently show an overview of all entries including `thumbnails`.

These `thumbnails` are small (200 pixels wide), yet we always load the full size `image` from the database. This is not particularly `efficient`, since all these (potentially very large) `digital images` need to be transferred to the client. We will try to fix this in this exercise.

We provide two new `Python files` with this exercise (`ImageManip.py` and `ImageHelper.py`). The first provides some basic `image processing` techniques (from last week). The latter provides functionality to create Pillow `images` from `binary data` (and vice versa) or to load Pillow `images` from a `URL`.

Familiarize yourself with the two files. You do not need to understand everything in the `Python code`, but make sure that you read the comments and that you understand what kind of functionality is given.

Now perform the following tasks:

1. In the `BuildDB.py` script, import the two provided files and Pillow:

```
import ImageHelper
import ImageManip
from PIL import Image
```

2. In the `BuildDB.py` script add one more column to the database called `Thumbnail` of type `BLOB`. This will store our thumbnail.

3. Adapt the `addImage` function, such that it creates a Pillow `image` from the `imageData` variable (look in the `ImageHelper` file for a function you can use for this task). Create the `thumbnail` (see file `ImageManip`). Then convert the `image` back to a binary blob and store it in the `Thumbnail` field of our database.

See the comments in the `BuildDB.py` file for more details.

4. In the `Server.py` script add a new route `/thumbnail/<id:int>`. This should be exactly the same as the `/imageraw/<id:int>` route (which already exists), with one exception: It should return the `Thumbnail` instead of the `Content` field.
5. Lastly, in the `Index.tpl` make sure, that your new `/thumbnail` route is used instead of the `/imageraw`. On the details page the original sized `image` should stay of course.

Problem 4.4 (Displaying Annotations)

In this exercise we will finally give our database frontend the ability to display annotations on top of our `images`. For now, these annotations come from files already provided in the `Kirmes` repository in the `xml/` subfolder. Each of the files in this directory describes areas (rectangles) in a given `image`, along with a description text.

We have prepared the `parsing` of these files for you, so you don't need to change anything in the `BuildDB.py` script. Nevertheless, check the table creation near the end of the file (from line 246). In addition to the `Images` table we worked with for the last couple of weeks, we now have a second table in our database, called `Annotations`. This table stores the following information:

1. `Id`: The id of the annotation (analogous to the `Id` field in the `Images` table).
2. `ImageId`: The id of the annotated `digital image`.
3. `Description`: A text describing the annotations.
4. `X, Y, Width, Height`: The position and dimensions of the rectangle in the `digital image`.

The `ImageId` is a `foreign key`, which references the `primary key` `Id` attribute of the `Images` table. For example, an annotation entry with `ImageId=27` defines an annotation for the image entry with `Id=27`. Note, that multiple annotations might reference the same `digital image`.

You don't need to do anything in this file, but make sure that you run it, so that your database is filled with the annotation data. Double check in the `DB Browser`, that the `Annotation` table is properly created and filled.

Now our frontend just needs to display the annotation information. To this end, amend the `/details/` route in the `Server.py` script, such that for the given image id, it `queries` the `database` for annotations.

In the `Details.tpl` file, iterate over the annotations (if any exist), and create a `<rect>` and a `<text>` for each. Fill in the information from the annotation (position and size of the rectangle, description for the text). See the course notes for details, if you are unsure how this works.

Check if everything works as expected by visiting the `/details/` page for an `image`, which has annotations. Not too many `images` actually have annotations, but some do. For example the `image` with id 146 should have a couple.

Make sure that by hovering the mouse over an annotation region, the rectangle highlights (gets brighter) and the description text is shown.

We will now give the user the ability to edit annotations directly in the browser. The idea is that changing the values of an annotation (position, size, text) is always easier in a graphical user interface than by typing in the values in an XML file.

The process requires two parts. First the user must be able to `interactively` change the values in the browser. Second, the changes they made must be saved back to the `database`.

In order to ensure a pleasant user experience the first part should be performed directly in the browser, so that not every mouse click must be sent to the server and back. Since this requires JavaScript, we have provided this part for you.

Run your server and visit a details page of any `image`, which has annotations, e.g. `http://localhost:8080/details/146`. At the bottom you should see a checkbox `Edit Annotations`. If this is checked, you should see a list of all annotations.

The currently selected element in this list is editable. You can change the annotation description in the text box. You can change the position and size of the annotation rectangle by dragging the marked (red) rectangle in the `image`. Note that you can both move and resize the rectangle.

New annotations can be added with the `New Annotation` button at the bottom and deleted by clicking the bin icon.

The changes you made are sent to the server, when the `Save Changes` button is clicked. Saving the changes in the `database` is for you to `implement`.

Right now clicking `Save Changes` should do nothing (even though the website displays a notification saying that the changes have been saved).

You can verify that saving is not working by making some changes. Then click `Save Changes` and refresh the page. All changes should be gone (because they are not stored in the database).

Problem 4.5 (Editing Annotations)

In the `Server.py` script you can find a new route `/edit_annotations`. Since this receives data (i.e. the changes you made to the annotations), it is marked as `POST`.

The function loops over a list of changes and gets the necessary data.

`Implement` the following: For each entry in the list of changes, issue the correct SQL command to update the values (hint: `UPDATE ...`). At the end of the function, commit your changes to the database (`db.commit()`).

Test your function! In the browser, edit one or multiple annotations and click `Save Changes`. Refresh the page. Your changes should still be there!

Problem 4.6 (Deleting Annotations)

Complete the `/edit_annotations` route by issuing a `DELETE` command for each entry passed to this function. Again, don't forget to commit your changes.

Test your code by deleting entries in the browser and refreshing the page!

Problem 4.7 (Adding Annotations)

Adding new annotations (`/new_annotations`) is slightly more complicated (but not much). Note that this function takes in the `imageID` as an argument.

In the loop, extract the individual fields from the `annotation` variable (similar to the way it's done in `/edit_annotations`). Since this is a `new` annotation, there is no `annotationID` this time.

Issue an `INSERT` command for each new annotation. Then get the id of the newly stored entry (`cursor.lastrowid`) and append this id to the `newIds` list. These new ids will be sent back to the client (browser) at the end of the function. This is already `implemented`.

Lastly, test your functionality! You should now be able to add new annotations in the browser, which will persist even if you refresh the page.

Chapter 12

Ontologies, Semantic Web for Cultural Heritage


In the last chapter IWGS, we will discuss a virtual research environment for [cultural heritage](#). Before we present the system itself, we take a close look at the underlying technology: ontologies, semantic web technologies, and linked open data.

12.1 Documenting our Cultural Heritage


Before we even start talking about the [WissKI](#) system, we should become clear on the concepts involved. We start out with the notion of [cultural heritage](#) itself.

Documenting our Cultural Heritage

- ▷ **Definition 12.1.1.** **Cultural heritage** is the legacy of physical artifacts **cultural artefacts** and practices, representations, expressions, **knowledge**, or skills – **intangible cultural heritage (ICH)** of a group or society that is inherited from past generations.
- ▷ **Problem:** How can we understand, conserve, and learn from our **cultural heritage**?
- ▷ **Traditional Answer:** We collect **cultural artefacts**, study them carefully, relate them to other **artefacts**, discuss the findings, and publish the results. We display the **artefacts** in museums and galleries, and educate the next generation.
- ▷ **DigHumS Answer:** In “Digital Humanities and Social Sciences”, we want to represent our **cultural heritage** digitally, and utilize computational tools to do so.
- ▷ **Practical Question:** What are the best representation formats and tools?

Michael Kohlhase: Inf. Werkzeuge © G/SW 2

347

2024-02-08 

There is another context in which we want to understand the [WissKI](#) system: that of [research data](#). We will introduce the basic concepts now.

Research Data in a Nutshell

- ▷ **Definition 12.1.2.** **Research data** is any **information** that has been collected, observed, generated or created to validate original research findings. Although usually digital, research data also includes non-digital formats such as laboratory notebooks

and diaries.

▷ **Types of research data:**

- ▷ documents, spreadsheets, laboratory notebooks, field notebooks, diaries,
- ▷ questionnaires, transcripts, codebooks, test responses,
- ▷ audiotapes, videotapes, photographs, films,
- ▷ **cultural artefacts**, specimens, samples,
- ▷ data files, database contents (video, audio, text, images), digital outputs,
- ▷ models, algorithms, scripts,
- ▷ contents of an application (input, output, logfiles, schemata),
- ▷ methodologies and workflows, standard operating procedures, and protocols,

- ▷ **Non-digital Research Data** such as **cultural artefacts**, laboratory notebooks, ice-core samples, or sketchbooks is often unique. Materials could be digitized, but this may not be possible for all types of **data**.

The very idea of **research data** is they are retained to justify the published research: in particular just publishing tables of results and experiment descriptions in journals is not enough.

In the past, this has led to the practice of keeping meticulous lab books in the experimental sciences, and in recent times to the practice of publishing original data together with the results, so that experiments can be replicated and derived results can be re-calculated. This being pushed through the scientific organizations in the last decades.

But publishing raw data is also insufficient: experiments can only be replicated and derivations can only be checked if the underlying data can be obtained in practice, are complete and correct, and can be interpreted by the reader. This led to substantial institutional attention and – consequently – to many new developments:

FAIR Research Data: The Next Big Thing

- ▷ **Principle:** Scientific experiments must be replicated, and derivations must be checkable to be trustworthy. (consensus of scientific community)
- ▷ **Intuition:** **Research data** must be retained for justification, shared for synergies!
- ▷ **Consequence:** Virtually all scientific funding agencies now require some kind of **research data** strategy in proposals. (tendency: getting stricter)
- ▷ **Problem:** Not all forms of **data** are actually useable in practice.
- ▷ **Definition 12.1.3 (Gold Standard Criteria).** **Research data** should be **FAIR**:
 - ▷ **Findable:** easy to identify and find for both humans and computers, e.g. with metadata that facilitate searching for specific datasets,
 - ▷ **Accessible:** stored for long term so that they can easily be accessed and/or downloaded with well-defined access conditions, whether at the level of metadata, or at the level of the actual data,
 - ▷ **Interoperable:** ready to be combined with other datasets by humans or computers, without ambiguities in the meanings of terms and values,

- ▷ **Reusable:** ready to be used for future research and to be further processed using computational methods.

Consensus in the [research data](#) community; for details see [FAIR18; Wil+16].

- ▷ **Open Question:** How can we achieve FAIR-ness in a discipline in practice?

After these general considerations about [research data](#), let us come back our primary concern in IWGS: [research data](#) in the humanities and social sciences.

If we look at the categories of [research data](#) we can expect in the humanities and social sciences, then we can categorize them into four broad categories. And we can see that we have already learned about many of them in IWGS.

Categories of Data in DigiHumS and their Formats

- ▷ We distinguish four broad categories of [data](#) in DigiHumS.
- ▷ **Definition 12.1.4. Concrete data:** digital representations of [artefacts](#) in terms of simple data,
 - ▷ e.g. [raster images](#) as pixel arrays in JPEG. (see chapter 11)
 - ▷ e.g. books identified by author/title/publisher/pubyear. (see chapter 9)
- ▷ **Definition 12.1.5. Narrative data:** documents and text fragments used for communicating [knowledge](#) to humans.
 - ▷ e.g. [plain text](#) and [formatted text](#) with [markup code](#) (see chapter 4 (Documents as Digital Objects) in the IWGS lecture notes)
- ▷ **Definition 12.1.6. Symbolic data:** descriptions of object and facts in a formal language
 - ▷ e.g. 3+5 in [Python](#) (see chapter 2 (Introduction to Programming) in the IWGS lecture notes)
- ▷ **Definition 12.1.7. Metadata:** “data about data”, e.g. who has created these facts, [images](#), or documents, how do they relate to each other? (not covered yet)
- ▷ **Observation 12.1.8. Metadata** are the resources, DigiHumS results are made of (↪ [support that](#))
The other categories digitize [artefacts](#) and [auxiliary data](#).
- ▷ **Observation 12.1.9.** We will need all of these – and their combinations – to do DigiHumS.

The last kind – [metadata](#) – is arguably the most important kind in the it concerns the relations between [artefacts](#), which are usually digitized into [concrete data](#).

WissKI: a Virtual Research Env. for Cultural Heritage

- ▷ **Definition 12.1.10. WissKI** is a virtual research environment (VRE) for managing

scholarly data and documenting **cultural heritage**.

- ▷ **Requirements:** For a virtual research environment for **cultural heritage**, we need
 - ▷ scientific communication about and documentation of the **cultural heritage**
 - ▷ networking **knowledge** from different disciplines (**transdisciplinarity**)
 - ▷ high-quality data acquisition and analysis
 - ▷ safeguarding authorship, authenticity, persistence
 - ▷ support of scientific publication
- ▷ **WissKI** was developed by the research group of Prof. Günther Görtz at FAU Erlangen-Nürnberg and is now used in hundreds of DH projects across Germany.
- ▷ FAU supports **cultural heritage** research by providing hosted **WissKI** instances.
 - ▷ See <https://wisski.data.fau.de> for details
 - ▷ We will use an instance for the Kirmes paintings in the homework assignments

This leads to the following plan for the rest of the chapter.

Documenting Cultural Heritage: Current State/Preview

- ▷ Pre-DH State of **cultural heritage** documentation:
 - ▷ **scientific communication/documentation** by journal articles/books
 - ▷ **persistence**: paper records, file cards, **databases** (**like our KirmesDB**)
 - ▷ **Analysis**: manual examination of **artefacts** in museums/archives.
- ▷ **Idea**: Use more technology to do better.
- ▷ **Preview**: **WissKI** uses **semantic web** technologies to do just that. We will now
 - ▷ Motivate the **semantic web** (**why do we need more than the WWW**)
 - ▷ introduce ontologies, linked open data and their technology stacks
 - ▷ show off **WissKI** and offer a little project based on Kirmes corpus.

12.2 Systems for Documenting the Cultural Heritage

Let us now have a look at how we can use digital systems to document the **cultural heritage**. This is the backdrop against which we need to position the **WissKI** system. The traditional methods of documenting **cultural artefacts** is in form of often handwritten – ledgers that inventory the collections of museums.

Documenting Cultural Artefacts: Inventory Books

- ▷ **Definition 12.2.1.** An **inventory book** is a ledger that identifies, describes, and records provenance of the **artefacts** in the collection of a museum.

▷ Example 12.2.2 (An Inventory Book).

INVENTAR JAHR NR.	KÜNSTLER	GEGENSTAND, BESCHREIBUNG, BEZEICHNUNG	TECHNIK, WERKSTOFF	MAASSE	ERWERBUNG	ANKAUFSPREIS	SCHENKUNGSPREIS	BEMERKUNGEN
1899/19	Polissier	Ringförmiges	Tausche nach	14 250	aus dem			
85	Fischerlein							
16	Folius							
17	Kallmann							
88	Dick							
87	W. S. S.							

▷ Problems: non-digital, only single-user access, institution-local, no querying, ...

If we want to improve on – or just digitize **inventory books**, the most obvious idea at least with what we have learned in IWGS – is to put the data into a **database** for persistence and use a **web application** for the user interface. Instead of surveying the multitude existing systems we want to improve on, let us briefly show an example.

Cultural Artefacts in Databases: Example

▷ Example 12.2.3. A typical database for cultural artefacts: (HiDa/MIDAS)

HiDa/MIDAS-Datenbank
Projekt zur Nürnberger Goldschmiedekunst


The system we see above is an instance of the HiDa/MIDAS system, which is in use in many museums for managing their collections. HiDa [HiDa] is a conventional (and commercial) rela-

tional database with a sophisticated user interface for data acquisition, reporting, exporting, and publication. Database schemata can be chosen from a set of options; here we see the MIDAS schema [BHK16].

The HiDa/MIDAS system is by no means the only one on the market, but the architecture is typical for the state of the art in most cultural institutions worldwide.

Cultural Artefacts in Databases: Pro/Con


- ▷ **Databases of Cultural Artefacts – Advantages:**
 - ▷ persistence, multi-user access, structured data,
 - ▷ web/catalog publication, standardized exports,
 - ▷ standardized performant query language.
- ▷ **Databases of Cultural Artefacts – Problems:**
 - ▷ identifiers are database local ~> no trans database relations,
 - ▷ database schemata are inflexible <-> we need extensions in practice,
 - ▷ free text as an un-structured, untapped resource.
- ▷ **Idea:** Relational databases impose structure, let's try something very unstructured: the world wide web. (up next)



Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

355

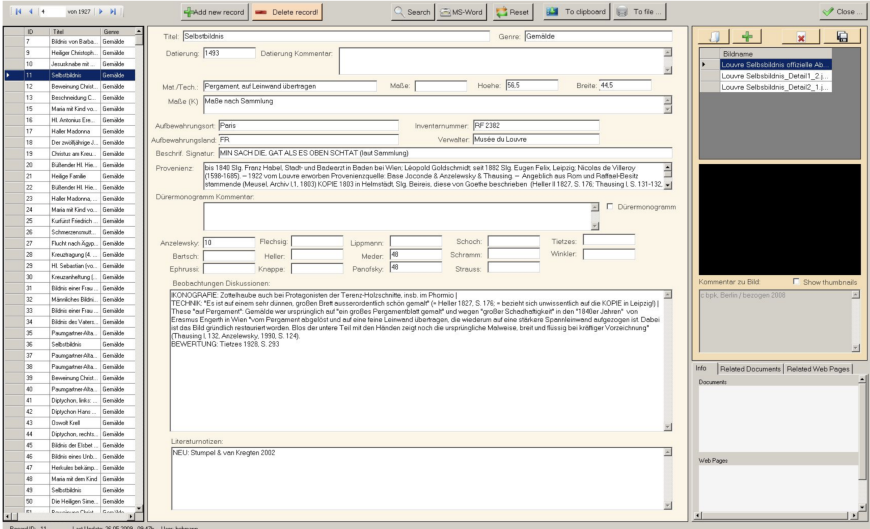
2024-02-08




Here is another example.

Cultural Artefacts in Databases II

▷ **Example 12.2.4.** Another database for cultural artefacts:






Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

356

2024-02-08



Let us see whether this idea has merit.

Using the Web for the Cultural Heritage

- ▷ **Idea:** Why not use the [world wide web](#) as a tool?
 - ▷ it is inherently distributed and networked,
 - ▷ the data formats [HTML](#) and [XML](#) are highly flexible,
 - ▷ gives us instantaneous access to information/images/... ,
 - ▷ allows collaboration and discussion. (wikis, fora, blogs)

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 357 2024-02-08

Again, an example is in order to help understand the issues at hand.

Cultural Artefacts on the Web

- ▷ **Example 12.2.5.** A text about a [cultural artefact](#) (an etching by Dürer)



WIKIPEDIA The Free Encyclopedia

Not logged in [Talk](#) [Contributions](#) [Create account](#) [Log in](#)

Article [Talk](#) [Read](#) [Edit](#) [View history](#)

Melencolia I

From Wikipedia, the free encyclopedia

Melencolia I is a 1514 engraving by the German Renaissance artist **Albrecht Dürer**. The print's central subject is an enigmatic and gloomy winged female figure thought to be a [personification of melancholia](#). Holding her head in her hand, she stares past the busy scene in front of her. The area is strewn with symbols and tools associated with craft and carpentry, including an [hourglass](#), [weighing scales](#), a [hand plane](#), a [claw hammer](#), and a [saw](#). Other objects relate to alchemy, geometry or numerology. Behind the figure is a structure with an embedded [magic square](#), and a ladder leading beyond the frame. The sky contains a rainbow, a comet or planet, and a bat-like creature bearing the text that has become the print's title.

Dürer's engraving is one of the most well-known extant [old master prints](#), but, despite a vast art-historical literature, it has resisted any definitive interpretation. Dürer may have associated melancholia with creative activity;^[2] the woman may be a representation of a [Muse](#), awaiting inspiration but fearful that it will not return. As such, Dürer may have intended the print as a veiled self-portrait. Other art historians see the figure as pondering the nature of beauty or the value of artistic creativity in light of rationalism,^[3] or as a purposely obscure work that highlights the limitations of [allegorical](#) or symbolic art.

The art historian [Erwin Panofsky](#), whose writing on the print has received the

Melencolia I ^[1] (with annotations)

Artist	Albrecht Dürer
Year	1514
Type	engraving
Dimensions	24 cm × 18.8 cm (9.4 in × 7.4 in)

Print/export

▷ **Question:** Just how does the etching discussed here relate to Albrecht Dürer?

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 358 2024-02-08

We collect the properties of the various approaches to documenting [cultural artefacts](#) to see how to proceed.

Using the Web for Cultural Heritage

- ▷ **Problems:** with using the [Web](#) as a resource
 - ▷ Information is often of dubious quality (imprecise, typos, incomplete, ...)
 - ▷ Information is primarily written for human consumption
 - ▷ ~> not machine-actionable, but full text search works (e.g. Google)
 - ▷ sometimes we can use established structures (e.g. Infobox in Wikipedia)

- ▷ **Evaluation:** The **web** is complementary to **databases** on the structure-vs-flexibility tradeoff scale for **cultural heritage** systems. (we need both)
- ▷ **Idea:** Use the **semantic web** for **cultural heritage**
 - ▷ **Goal:** Make information accessible for humans and machines
 - ▷ meaning capture by reference to real-world objects
 - ▷ globally unique identifiers of **cultural artefacts** (\cong URIs)
 - ▷ inference (get out more than you put in!)

12.3 The Semantic Web

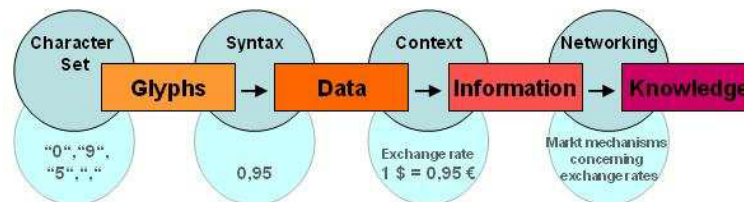
In this section we will introduce the **semantic web**. That tries to transform the **World Wide Web** from a human understandable web of **multimedia** documents into a “web of machine understandable data”. In this context, “machine-understandable” means that machines can draw inferences from data they have access to, so that they can make use of the **knowledge** that is implicit – i.e. not explicitly stated, but can be derived from other information (by humans) – in the web.

We will now define the term **semantic web** and discuss the pertinent ideas involved. There are two central ones, we will cover here:

- Information and data come in different levels of explicitness; this is usually visualized by a “ladder” of information.
- if information is sufficiently machine-understandable, then we can automate drawing conclusions.

The Semantic Web

- ▷ **Definition 12.3.1.** The **semantic web** is the result including of semantic content in **web pages** with the aim of converting the **WWW** into a machine-understandable “web of data”, where **inference** based services can add value to the ecosystem.
- ▷ **Idea:** Move web content up the ladder, use **inference** to make connections.



- ▷ **Example 12.3.2.** Information not explicitly represented (in one place)

Query: *Who was US president when Barak Obama was born?*

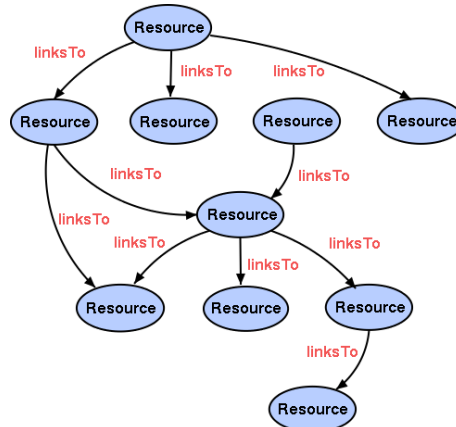
Google: ... *BIRTH DATE: August 04, 1961...*

Query: *Who was US president in 1961?*

Google: *President: Dwight D. Eisenhower [...] John F. Kennedy (starting Jan. 20.)*

Humans understand the text and combine the information to get the answer. Machines need more than just text \rightsquigarrow **semantic web** technology.

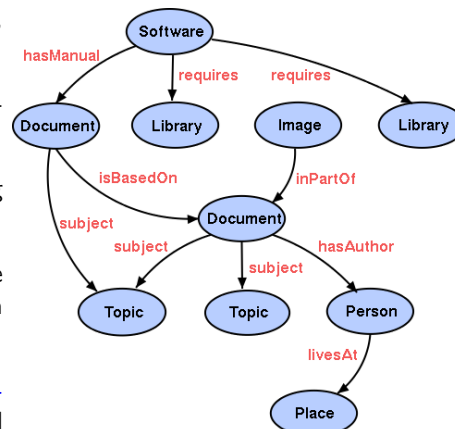
- ▷ **Resources:** identified by URIs, untyped
- ▷ **Links:** href, src, ... limited, non-descriptive
- ▷ **User:** Exciting world - semantics of the resource, however, gleaned from content
- ▷ **Machine:** Very little information available - significance of the links only evident from the context around the anchor.



Let us now contrast this with the envisioned [semantic web](#).

The Semantic Web

- ▷ **Resources:** Globally identified by URIs or Locally scoped (Blank), Extensible, Relational.
- ▷ **Links:** Identified by URIs, Extensible, Relational.
- ▷ **User:** Even more exciting world, richer user experience.
- ▷ **Machine:** More processable information is available (Data Web).
- ▷ **Computers and people:** Work, learn and exchange knowledge effectively.



Essentially, to make the web more machine-processable, we need to classify the resources by the concepts they represent and give the links a [meaning](#) in a way, that we can do inference with that. The ideas presented here gave rise to a set of technologies jointly called the “semantic web”, which we will now summarize before we return to our logical investigations of knowledge representation techniques.

Towards a “Machine-Actionable Web”

- ▷ **Recall:** We need external agreement on [meaning](#) of annotation tags.
- ▷ **Idea:** standardize them in a community process (e.g. DIN or ISO)
- ▷ **Problem:** Inflexible, Limited number of things can be expressed
- ▷ **Better:** Use [ontologies](#) to specify [meaning](#) of annotations

- ▷ Ontologies provide a vocabulary of terms
- ▷ New terms can be formed by combining existing ones
- ▷ **Meaning (semantics)** of such terms is formally specified
- ▷ Can also specify relationships between terms in multiple ontologies
- ▷ Inference with annotations and ontologies (get out more than you put in!)
- ▷ Standardize annotations in **RDF** [KC04] or **RDFa** [Her+13b] and ontologies on **OWL** [OWL09]
- ▷ Harvest **RDF** and **RDFa** in to a **triplestore** or **OWL** reasoner.
- ▷ **Query** that for implied knowledge (e.g. **chaining multiple facts from Wikipedia**)
SPARQL: Who was US President when Barack Obama was Born?
DBpedia: John F. Kennedy (was president in August 1961)

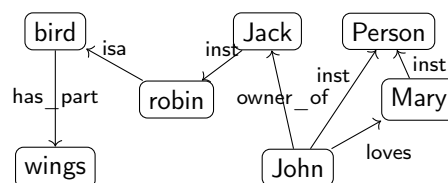
12.4 Semantic Networks and Ontologies

To get a feeling for ontologies and how they enable the “machine-actionable web” and how that helps us in DH, we take a look at “semantic networks”, which are an early form of ontologies. They allow us to explain many of the basic functionalities of the “semantic web” without getting too much into details of the technologies involved. We will preview that at the end of this section and go into details section 12.6.

Semantic networks are a very simple way of arranging knowledge about **objects** and **concepts** and their relationships in a **graph**.

Semantic Networks [CQ69]

- ▷ **Definition 12.4.1.** A **semantic network** is a **directed graph** for representing knowledge:
 - ▷ **nodes** represent **objects** and **concepts** (classes of **objects**)
(e.g. **John (object)** and **bird (concept)**)
 - ▷ **edges** (called **links**) represent relations between these (isa, father_of, belongs_to)
- ▷ **Example 12.4.2.** A **semantic network** for birds and persons:

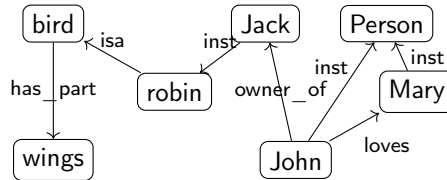


- ▷ **Problem:** How do we derive new information from such a network?
- ▷ **Idea:** Encode taxonomic information about **objects** and **concepts** in special **links** (“isa” and “inst”) and specify property inheritance along them in the process model.

Even though the [network](#) in Example 12.4.2 is very intuitive (we immediately understand the [concepts](#) depicted), it is unclear how we (and more importantly a machine that does not associate [meaning](#) with the labels of the [nodes](#) and [edges](#)) can draw [inferences](#) from the “knowledge” represented.

Deriving Knowledge Implicit in Semantic Networks

- ▷ **Observation 12.4.3.** *There is more knowledge in a [semantic network](#) than is explicitly written down.*
- ▷ **Example 12.4.4.** In the network below, we “know” that *robins have wings* and in particular, *Jack has wings*.



- ▷ **Idea:** Links labeled with “isa” and “inst” are special: they propagate properties encoded by other links.
- ▷ **Definition 12.4.5.** We call links labeled by
 - ▷ “isa” an **inclusion** or **isa link** (inclusion of concepts)
 - ▷ “inst” **instance** or **inst link** (concept membership)

We now make the idea of “propagating properties” rigorous by defining the notion of [derived relations](#), i.e. the relations that are left implicit in the network, but can be added without changing its [meaning](#).

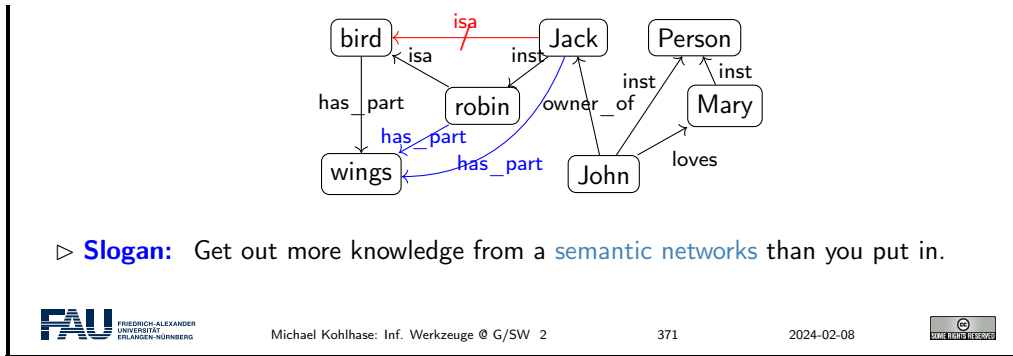
Deriving Knowledge Semantic Networks

- ▷ **Definition 12.4.6 (Inference in Semantic Networks).** We call all link labels except “inst” and “isa” in a [semantic network](#) **relations**.

Let N be a [semantic network](#) and R a [relation](#) in N such that $A \xrightarrow{\text{isa}} B \xrightarrow{R} C$ or $A \xrightarrow{\text{inst}} B \xrightarrow{R} C$, then we can **derive** a [relation](#) $A \xrightarrow{R} C$ in N .

The process of [deriving](#) new [concepts](#) and [relations](#) from existing ones is called **inference** and [concepts/relations](#) that are only available via **inference implicit** (in a [semantic network](#)).

- ▷ **Intuition:** [Derived relations](#) represent knowledge that is implicit in the network; they could be added, but usually are not to avoid clutter.
- ▷ **Example 12.4.7.** [Derived relations](#) in Example 12.4.4



Note that Definition 12.4.6 does not quite allow to derive that *Jack is a bird* (did you spot that “isa” is not a relation that can be inferred?), even though we know it is true in the world. This shows us that inference in semantic networks has to be very carefully defined and may not be “complete”, i.e. there are things that are true in the real world that our inference procedure does not capture.

Dually, if we are not careful, then the inference procedure might derive properties that are not true in the real world even if all the properties explicitly put into the network are. We call such an inference procedure *unsound* or *incorrect*.

These are two general phenomena we have to keep an eye on.

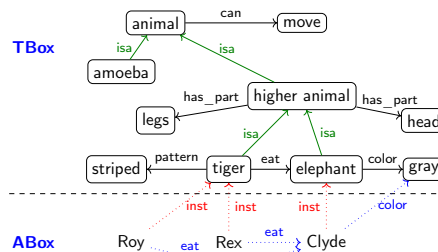
Another problem is that semantic networks (e.g. in Example 12.4.2) confuse two kinds of concepts: individuals (represented by proper names like *John* and *Jack*) and concepts (nouns like *robin* and *bird*). Even though the isa and inst link already acknowledge this distinction, the “has_part” and “loves” relations are at different levels entirely, but not distinguished in the networks.

Terminologies and Assertions

▷ Remark 12.4.8. We should distinguish concepts from objects.

▷ Definition 12.4.9. We call the subgraph of a semantic network *N* spanned by the isa links and relations between concepts the terminology (or TBox, or the famous Isa Hierarchy) and the subgraph spanned by the inst links and relations between objects, the assertions (or ABox) of *N*.

▷ Example 12.4.10. In this semantic network we keep objects concept apart notationally:



In particular we have objects “Rex”, “Roy”, and “Clyde”, which have (derived) relations (e.g. *Clyde is gray*).

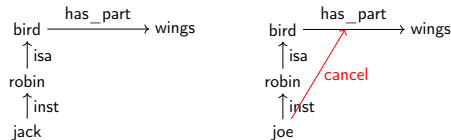
But there are severe shortcomings of semantic networks: the suggestive shape and node names give (humans) a false sense of meaning, and the inference rules are only given in the process model

(the [implementation](#) of the [semantic network](#) processing system).

This makes it very difficult to assess the strength of the [inference system](#) and make assertions e.g. about [completeness](#).

Limitations of Semantic Networks

- ▷ What is the [meaning](#) of a [link](#)?
 - ▷ [link](#) labels are very suggestive (misleading for humans)
 - ▷ [meaning](#) of [link](#) types defined in the process model (no denotational semantics)
- ▷ **Problem:** No distinction of optional and defining traits!
- ▷ **Example 12.4.11.** Consider a robin that has lost its wings in an accident:

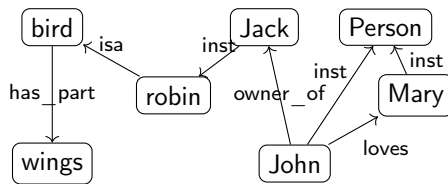


“Cancel-links” have been proposed, but their status and process model are debatable.

To alleviate the perceived drawbacks of semantic networks, we can contemplate another notation that is more linear and thus more easily [implemented](#): function/argument notation.

Another Notation for Semantic Networks

- ▷ **Definition 12.4.12.** [Function/argument notation](#) for [semantic networks](#)
 - ▷ interprets [nodes](#) as arguments (reification to individuals)
 - ▷ interprets [links](#) as functions (predicates actually)
- ▷ **Example 12.4.13.**



```
isa(robin,bird)
haspart(bird,wings)
inst(Jack,robin)
owner_of(John,robin)
loves(John,Mary)
```

- ▷ **Evaluation:**
 - + linear notation (equivalent, but better to implement on a computer)
 - + easy to give process model by deduction (e.g. in Prolog)
 - worse locality properties (networks are associative)

Indeed the function/argument notation is the immediate idea how one would naturally represent semantic networks for [implementation](#).

This notation has been also characterized as subject/predicate/object triples, alluding to simple

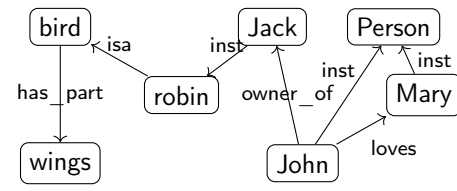
(English) sentences. This will play a role in the “semantic web” later. The next slide is a bit outside of the scope of IWGS, but we want to go into this anyway.

We have been talking about the “procedural model” of a [semantic network](#), which essentially specifies the [inference algorithm](#) that [derives](#) new [knowledge](#) in a network. There is an alternative to this: we can map the network language – [function/argument notation](#) for networks is an essential step for this – into a known language with an inference system. We call this kind of a mapping a “denotational semantics”, here into a language called first-order logic.

Building on the [function/argument notation](#) from above, we can now give a formal semantics for [semantic network](#): we translate them into [first-order logic](#) and use the semantics of that.

A Denotational Semantics for Semantic Networks

▷ **Observation:** If we handle *isa* and *inst* links specially in [function/argument notation](#)




robin ⊆ bird
 haspart(bird,wings)
 Jack ∈ robin
 owner_of(John, Jack)
 loves(John,Mary)

it looks like [first-order logic](#), if we take

- ▷ $a \in S$ to mean $S(a)$ for an [object](#) a and a [concept](#) S .
- ▷ $A \subseteq B$ to mean $\forall X.A(X) \Rightarrow B(X)$ and [concepts](#) A and B
- ▷ $R(A, B)$ to mean $\forall X.A(X) \Rightarrow (\exists Y.B(Y) \wedge R(X, Y))$ for a [relation](#) R .

▷ **Idea:** Take first-order deduction as process model (gives inheritance for free)




FRIEDRICH-ALEXANDER
 UNIVERSITÄT
 ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

375

2024-02-08



Indeed, the semantics induced by the translation to first-order logic, gives the intuitive [meaning](#) to the semantic networks. Note that this only holds only for the features of semantic networks that are representable in this way, e.g. the “cancel links” shown above are not (and that is a feature, not a [bug](#)).

But even more importantly, the translation to first-order logic gives a first process model: we can use first-order inference to compute the set of inferences that can be drawn from a semantic network.

Based on the intuitions from [semantic networks](#) we can now come to general ([semantic web](#)) ontologies.

What is an Ontology

▷ **Definition 12.4.14.** An [ontology](#) is a formal model of (an aspect of) the world. It

- ▷ introduces a [vocabulary](#) for the [objects](#), [concepts](#), and [relations](#) of a given [domain](#),
- ▷ specifies intended [meaning](#) of [vocabulary](#) in a [description logic](#) using
 - ▷ a set of [axioms](#) describing structure of the model
 - ▷ a set of [facts](#) describing some particular concrete situation

The [vocabulary](#) together with the collection of [axioms](#) is often called a [terminology](#) (or [TBox](#)) and the collection of facts an [ABox](#) ([assertions](#)).

In addition to the **represented axioms** and **facts**, the **description logic** determines a number of **derived** ones.

- ▷ **Definition 12.4.15.** A **vocabulary** often includes names for **classes** and **relationship** (also called **concepts**, and **properties**).
- ▷ *Remark 12.4.16.* If the **description logic** has a reasoner, we can automatically
 - ▷ detect **inconsistent axiom** systems
 - ▷ compute class membership and **taxonomies**.

There is a whole collection of standardized languages and interoperable systems that facilitate dealing with (very large) ontologies in practice. We will only give a summary preview here, leaving the detailed discussion to section 12.6.

Semantic Web Technology in a Nutshell

- ▷ **Ontologies** have become one of the standard devices for representing information about the **Web** and the world.
- ▷ **Definition 12.4.17.** This is facilitated and standardized by the :
 - ▷ **URIs** for representing **objects**,
 - ▷ **RDF triples** for representing **facts**,
 - ▷ **RDFa** for annotating **RDF triples** in **XML** documents,
 - ▷ **OWL** for representing **TBoxes**,
 - ▷ **triplestores** for storing (lots of) **RDF triples**,
 - ▷ **SPARQL** for **querying ontologies**,
 - ▷ **description logic reasoners** for deciding ontology consistency and concept subsumption,
 - ▷ **Protg** for authoring and maintaining **ontologies**,
- ▷ Details section 12.6.

Indeed, this list can be read as a technology roadmap for the **WissKI** system. We have already seen the most of the concepts in ??, we will discuss the technologies section 12.6, but first we will have a look at the **CIDOC CRM** ontology that is used in **WissKI**.

12.5 CIDOC CRM: An Ontology for Cultural Heritage

We have seen that **databases** are not the only choice for representing data about **cultural heritage**. Indeed, the **WissKI** system chooses **ontologies** as a basis for representation and **querying**.

To ensure interoperability, **WissKI** is based on the ISO-standardized **CIDOC CRM** ontology, which we will now introduce and explore.

Now, we can instantiate what we have learned about ontology-based information systems to **cultural heritage** disciplines. We collect all the bits and pieces and hint at the technologies (details section 12.6).

Ontologies for Cultural Artefacts

▷ **Idea:** Use **ontologies** for documenting cultural heritage.

- ▷ flexible schemata (OWL)
- ▷ easy data sharing
- ▷ open standards, free tools
- ▷ semantic querying via SPARQL

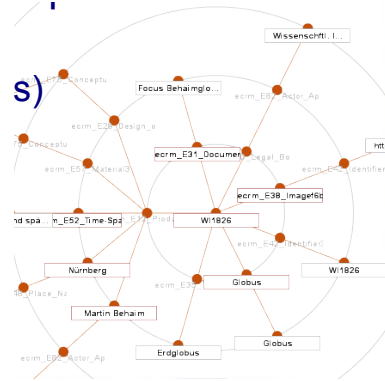
▷ **Idea:** We can use **RDF** like a Mindmap: **RDF** can

- ▷ represent relations between objects
- ▷ classify objects (web resources)

RDFa for document annotation

▷ Reference **ontologies** for interoperability:

- ▷ SUMO (Suggested Upper Model Ontology) [SUMO] for common knowledge,
- ▷ FOAF (Friend-of-a-Friend) [FOAF14] for persons and relations,
- ▷ **CIDOC CRM** for documentation of cultural heritage. (up next)



So let us look at the **CIDOC CRM** ontology in more detail. It has been developed by the Documentation Committee of the ICOM (International Council of Museums) over more than 20 years and has been standardized by the ISO. Even more importantly for our purposes here, the **CIDOC CRM** has been implemented in the **OWL** format, which gives us the use of the **semantic web technology stack**.

CIDOC CRM (Conceptual Reference Model)

▷ **Definition 12.5.1.** **CIDOC CRM** provides an extensible ontology for concepts and information in cultural heritage and museum documentation. It is the international standard (ISO 21127:2014) for the controlled exchange of **cultural heritage** information. The central classes include

- ▷ **space time** specified by title/identifier, place, era/period, time-span, and relationship to **persistent** items
- ▷ **events** specified by title/identifier, beginning/ending of existence, participants (people, either individually or in groups), creation/modification of things (physical or conceptual), and relationship to **persistent** items
- ▷ **material things** specified by title/identifier, place, the information object the material thing carries, part-of relationships, and relationship to **persistent** items
- ▷ **immaterial things** specified by title/identifier, information objects (propositional or symbolic), conceptual things, and part-of relationships

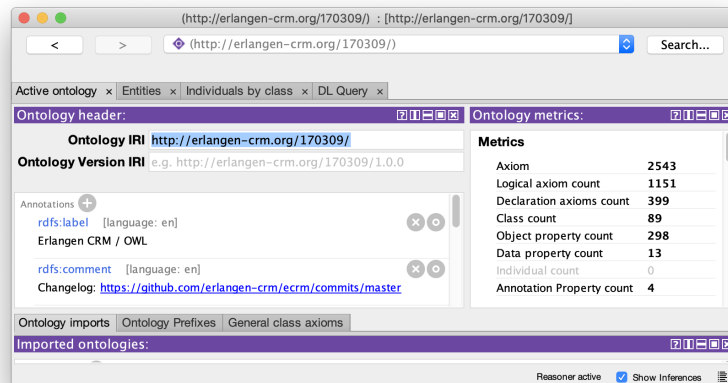
▷ **Definition 12.5.2.** **OWL** implements **CIDOC CRM** in **OWL**

- ▷ Details about [CIDOC CRM](#) can be found at [CC] and about [OWL](#) at [ECRMB; ECRMa].

One of the advantages of having [CIDOC CRM](#) in [OWL](#) is that we can use semantic web technologies to deal with it. Here we use one of the practically most important tools: [Protégé](#).

Protégé, an IDE for Ontology Development

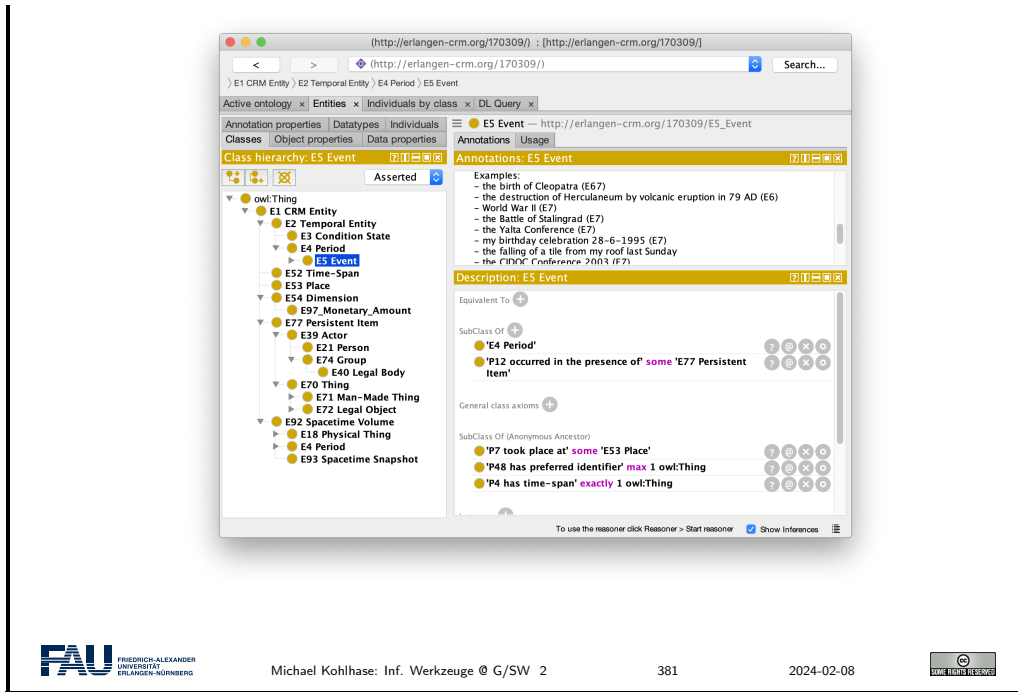
- ▷ **Definition 12.5.3.** [Protégé](#) [Pro] is an [integrated development environment](#) for [ontologies](#) represented in the [OWL](#) family. It comprises
 - ▷ a visual user interface for exploring and editing ontologies,
 - ▷ a [inference](#) component to ensure [ontology](#) consistency and minimality,
 - ▷ a facility for [querying](#) the loaded ontologies.
- ▷ **Example 12.5.4 (CIDOCCRM in Protégé).**



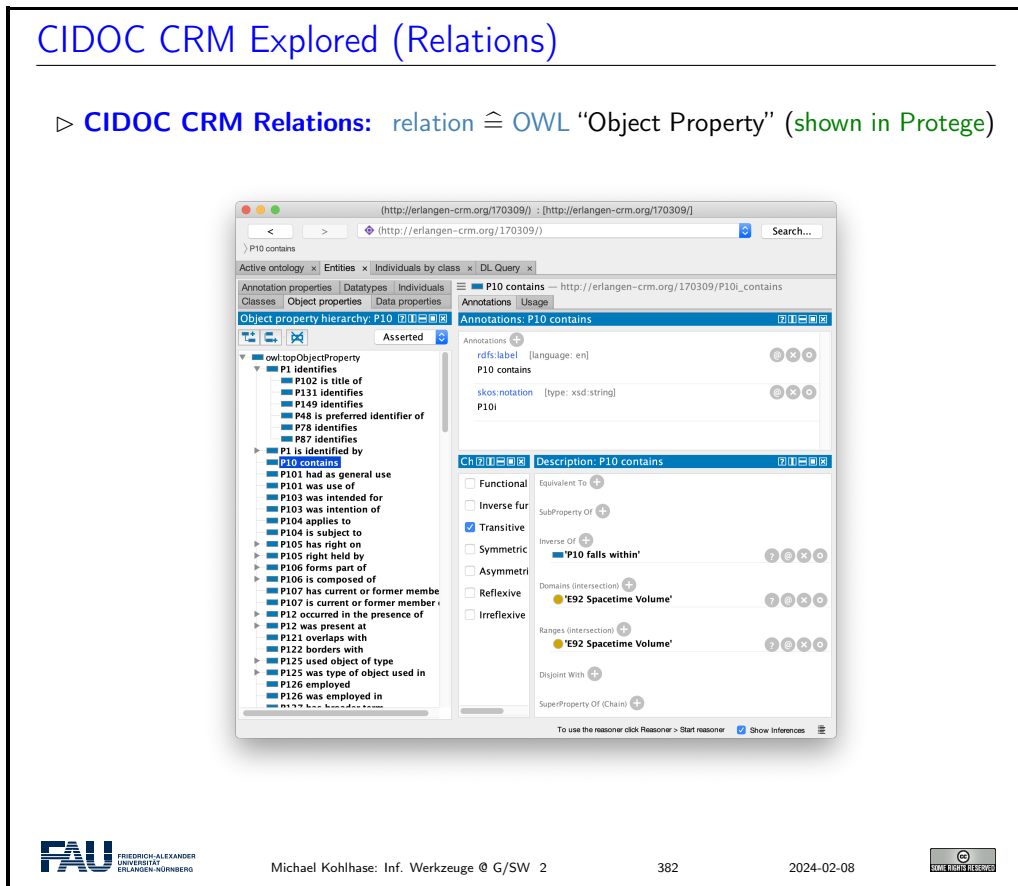
The backbone of the [CIDOC CRM](#) ontology is formed by the [concepts](#) (called “classes” in [OWL](#)). They form an inheritance hierarchy – of which the top part is shown on the left of the [Protégé window](#) below. The ontology provides – usually relatively abstract classes for all objects related to [cultural artefacts](#), their properties, and provenance.

CIDOC CRM Explored (Classes)

- ▷ **Idea:** Use semantic web technology to explore [OWL](#).
- ▷ **CIDOC CRM Classes:** $\text{concept} \hat{=} \text{OWL "Class"}$ (shown in [Protégé](#))

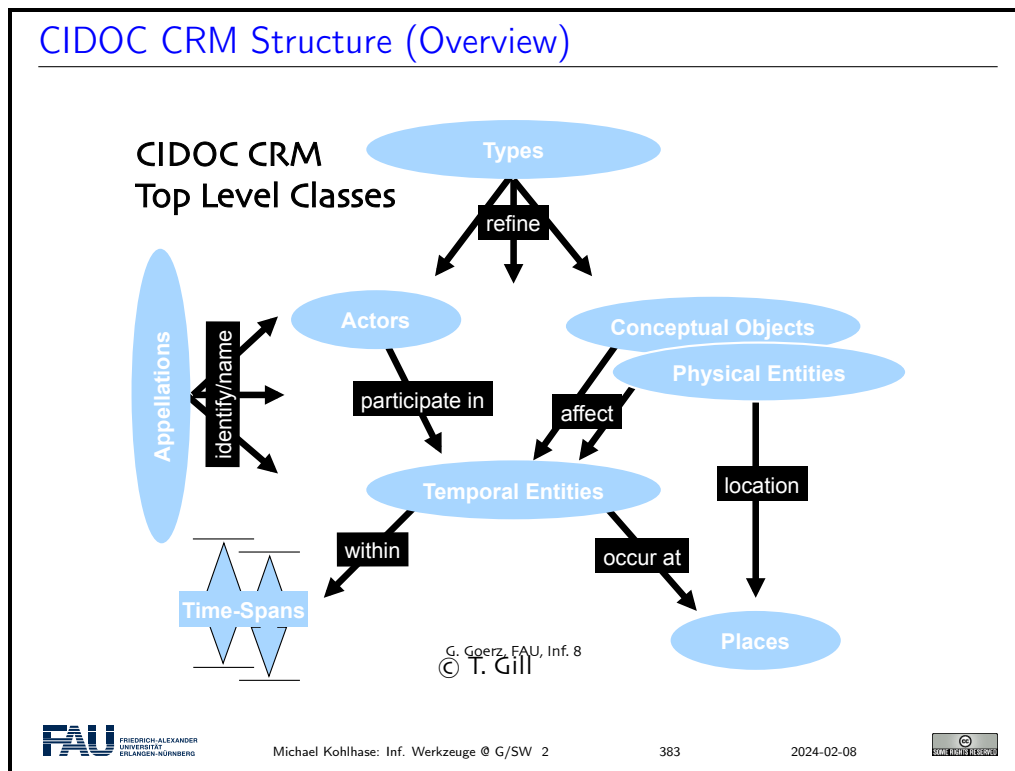


The concepts are complemented by the relations called “object properties” in OWL.



There are also a small number of “data properties”, i.e. properties whose values are concrete data like numbers, dates, or strings. They are less interesting structurally, but important in practice.

We can summarize the structure of the [CIDOC CRM ontology](#) in the following diagram.



Now that we understand the [CIDOC CRM ontology](#), we look into the process of modeling [cultural artefacts](#).

CIDOC-CRM Modeling

- ▷ **This is all good and dandy** but how do I concretely model [cultural artefacts](#)?
- ▷ **Answer:** CIDOC CRM is only a [TBox](#), we add an [ABox](#) of [objects](#) and [facts](#).
- ▷ **Example 12.5.5.** *Albrecht Dürer painted Melencolia 1 in Nürnberg*
We have two units of information here:
 1. Albrecht Dürer painted Melencolia 1
 2. this happened in the city of Nürnberg
- ▷ [CIDOC CRM](#) modeling decisions; we start with 1. *AD painted M 1*
 1. A painting *m* is an "Information Carrier" (E84)
 2. It was created in an "Production Event" *q* (E12)
 3. *m* is related to *q* via the "was produced by" relation (P108i)
 4. *q* was "carried out by" a "person" *d* (P14 E21)
 5. *d* "is identified by" an "actor appellation" *a* (P131 E82)
 6. *a* "has note" the string "Albrecht Dürer". (P3)
- ▷ [CIDOC CRM](#) modeling decisions; continuing with 2. *this happened in N*
 1. A painting *m* is an "Information Carrier" (E84)
 2. It was created in an "Production Event" *q* (E12)

3. m is related to q via the "produced by" relation	(P108i)
4. q "took place at" a "place" p	(P7 E53)
5. p "is identified by" a "place name" n	(P48 E3)
6. n "has note" the string "Nürnberg".	(P3)

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 384 2024-02-08

If we look more closely at the objects and relations Example 12.5.5, we see that

- a typical information unit results in a whole chain of objects connected by ontology relations
- parts of these chains are shared between information units

We address this now and introduce the concept of **ontology groups** and **ontology paths** for that.

CIDOC CRM Modelling (Ontology Paths)

▷ Modeling *Albrecht Dürer painted Melencolia 1 in Nürnberg* in CIDOC CRM

```

    graph LR
      m["m : E84"] -- P108i --> q["q : E12"]
      q -- P14 --> d["d : E21"]
      q -- P7 --> p["p : E53"]
      d -- P131 --> a["a : E82"]
      p -- P87 --> n["n : E48"]
      a -- P3 --> a_str["A. Dürer"]
      n -- P3 --> n_str["Nürnberg"]
    
```

Note that we need to create the intermediary **objects** q , d , a , and n .

▷ **Problem:** That is a lot of work for something very simple.

▷ **Definition 12.5.6.** We call sequence of facts $s_i \xrightarrow{p_i} o_i$, where $s_i = o_{i-1}$ an **ontology path** and any subtree an **ontology group**.

▷ **Problem Reformulated:** A simple statement like *Albrecht Dürer painted Melencolia 1* becomes a whole **ontology path** in CIDOC CRM.

▷ **But:** we can reuse intermediary **objects** and **facts**, and need fine grained models for flexibility.

▷ **Idea:** Maybe systems can take some of the pain out of modeling. (↷ WissKI)

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 385 2024-02-08

In Example 12.5.5, we have already seen one of the peculiarities of modeling complex situations in **ontologies**: the use of events as intermediate objects. This is a general phenomenon when modeling with ontologies, which we have to get used. to

Event-Oriented Modeling in CIDOC CRM

▷ **Observation 12.5.7.** *Ontologies make it easy to model facts with transitive verbs, e.g. Albrecht Dürer created Melencolia 1 (binary relation)*

▷ **Problem:** What about more complex situations with more arguments? E.g.

1. *Albrecht Dürer created Melencolia 1 with an etching needle (ternary)*

2. *Albrecht Dürer* created *Melencolia 1* with an etching needle in *Nürnberg* (four arguments)
 3. *Albrecht Dürer* created *Melencolia 1* with an etching needle in *Nürnberg* out of boredom (five)
- ▷ **Standard Solution:** Introduce “events” tied to the verb and describe those
 - ▷ **Example 12.5.8.** There was a creation event e with
 1. *Albrecht Dürer* as the agent,
 2. *Melencolia 1* as the product,
 3. *an etching needle* as the means,
 4. *boredom* as the reason,
 - ▷ **Consequence:** More than 1/3 of **CIDOC CRM** classes are events of some kind.

This “event-oriented” thinking is unfamiliar at first and takes practice to become natural. As a rule of thumb one should proceed as in the Melencolia example above. We first identify the “participants” in the situation, if these are more than two, we need to introduce an appropriate event (select from the ones provided by **CIDOC CRM**) and then connect the event to the object currently under consideration, and all the “participants” to the event.

12.6 The Semantic Web Technology Stack

In this section we discuss how we can apply **description logics** in the real world, in particular, as a conceptual and **algorithmic** basis of the **semantic web**. That tries to transform the **World Wide Web** from a human-understandable web of **multimedia** documents into a “web of machine-understandable data”. In this context, “machine-understandable” means that **machines** can draw **inferences** from **data** they have access to. Note that the discussion in this digression is not a full-blown introduction to **RDF** and **OWL**, we leave that to [SR14; Her+13a; Hit+12] and the respective **W3C** recommendations. Instead we introduce the ideas behind the mappings from a perspective of the description logics we have discussed above.

The most important component of the **semantic web** is a standardized language that can represent “data” about information on the **Web** in a machine-oriented way.

Resource Description Framework

- ▷ **Definition 12.6.1.** The **Resource Description Framework (RDF)** is a framework for describing resources on the web. It is an **XML** vocabulary developed by the **W3C**.
- ▷ **Note:** **RDF** is designed to be read and understood by **computers**, not to be displayed to people. (it shows)
- ▷ **Example 12.6.2.** **RDF** can be used for describing (all “objects on the **WWW**”)
 - ▷ properties for shopping items, such as price and availability
 - ▷ time schedules for web events
 - ▷ information about **web pages** (content, author, created and modified date)
 - ▷ content and rating for web pictures

- ▷ content for search engines
- ▷ electronic libraries

Note that all these examples have in common that they are about “objects on the [Web](#)”, which is an aspect we will come to now.

“Objects on the [Web](#)” are traditionally called “resources”, rather than defining them by their intrinsic properties – which would be ambitious and prone to change – we take an external property to define them: everything that has a [URI](#) is a web resource. This has repercussions on the design of [RDF](#).

Resources and URIs

- ▷ [RDF](#) describes resources with properties and property values.
- ▷ [RDF](#) uses Web identifiers ([URIs](#)) to identify resources.
- ▷ **Definition 12.6.3.** A [resource](#) is anything that can have a [URI](#), such as `http://www.fau.de`.
- ▷ **Definition 12.6.4.** A [property](#) is a resource that has a name, such as *author* or *homepage*, and a [property value](#) is the value of a property, such as *Michael Kohlhase* or `http://kwar.c.info/kohlhase`. (a [property value](#) can be another resource)
- ▷ **Definition 12.6.5.** A [RDF statement](#) s (also known as a [triple](#)) consists of a [resource](#) (the [subject](#) of s), a [property](#) (the [predicate](#) of s), and a [property value](#) (the [object](#) of s). A set of [RDF triples](#) is called an [RDF graph](#).
- ▷ **Example 12.6.6.** Statements: *[This slide]^{subj} has been [author]^{pred}ed by [Michael Kohlhase]^{obj}*

The crucial observation here is that if we map “subjects” and “objects” to “individuals”, and “predicates” to “relations”, the [RDF](#) triples are just relational ABox statements of description logics. As a consequence, the techniques we developed apply.

Note: Actually, a [RDF graph](#) is technically a [labeled multigraph](#), which allows multiple edges between any two nodes (the resources) and where nodes and edges are labeled by [URIs](#).

We now come to the concrete syntax of [RDF](#). This is a relatively conventional [XML](#) syntax that combines [RDF](#) statements with a common subject into a single “description” of that resource.

XML Syntax for RDF

- ▷ [RDF](#) is a concrete [XML](#) vocabulary for writing statements
- ▷ **Example 12.6.7.** The following [RDF](#) document could describe the slides as a resource

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description about="https://.../CompLog/kr/en/rdf.tex">
    <dc:creator>Michael Kohlhase</dc:creator>
```

```
<dc:source>http://www.w3schools.com/rdf</dc:source>
</rdf:Description>
</rdf:RDF>
```

This [RDF](#) document makes two statements:

- ▷ The subject of both is given in the about attribute of the `rdf:Description` element
 - ▷ The [predicates](#) are given by the element names of its [children](#)
 - ▷ The [objects](#) are given in the elements as [URIs](#) or [literal](#) content.
- ▷ **Intuitively:** [RDF](#) is a web-scalable way to write down [ABox](#) information.

Note that [XML](#) namespaces play a crucial role in using element to encode the [predicate URIs](#). Recall that an element name is a qualified name that consists of a [namespace URI](#) and a proper element name (without a colon character). Concatenating them gives a [URI](#) in our example the predicate [URI](#) induced by the `dc:creator` element is `http://purl.org/dc/elements/1.1/creator`. Note that as [URIs](#) go [RDF URIs](#) do not have to be [URLs](#), but this one is and it references (is redirected to) the relevant part of the Dublin Core elements specification [DCM12].

[RDF](#) was deliberately designed as a standoff markup format, where [URIs](#) are used to annotate web resources by pointing to them, so that it can be used to give information about web resources without having to change them. But this also creates maintenance problems, since web resources may change or be deleted without warning.

[RDFa](#) gives authors a way to embed [RDF](#) triples into web resources and make keeping [RDF](#) statements about them more in sync.

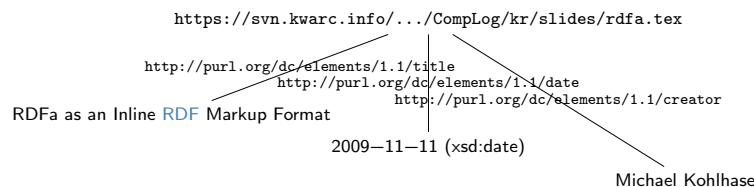
RDFa as an Inline RDF Markup Format

- ▷ **Problem:** [RDF](#) is a standoff markup format (annotate by [URIs](#) pointing into other files)

Definition 12.6.8. [RDFa](#) ([RDF](#) annotations) is a markup scheme for inline annotation (as [XML](#) attributes) of [RDF](#) triples.

- ▷ **Example 12.6.9.**

```
<div xmlns:dc="http://purl.org/dc/elements/1.1/" id="address">
  <h2 about="#address" property="dc:title">RDF as an Inline RDF Markup Format</h2>
  <h3 about="#address" property="dc:creator">Michael Kohlhase</h3>
  <em about="#address" property="dc:date" datatype="xsd:date"
    content="2009-11-11">November 11., 2009</em>
</div>
```



In the example above, the `about` and `property` attributes are reserved by [RDFa](#) and specify the subject and predicate of the [RDF](#) statement. The `object` consists of the body of the element,

unless otherwise specified e.g. by the content and datatype attributes for literals content. Let us now come back to the fact that RDF is just an XML syntax for ABox statements.

RDF as an ABox Language for the Semantic Web

- ▷ **Idea:** RDF triples are ABox entries $h R s$ or $h:\varphi$.
- ▷ **Example 12.6.10.** h is the resource for Ian Horrocks, s is the resource for Ulrike Sattler, R is the relation “hasColleague”, and φ is the class foaf:Person


```
<rdf:Description about="some.uri/person/ian_horrocks">
  <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  <hasColleague resource="some.uri/person/uli_sattler"/>
</rdf:Description>
```
- ▷ **Idea:** Now, we need a similar language for TBoxes (based on *ACC*)

In this situation, we want a standardized representation language for TBox information; OWL does just that: it standardizes a set of knowledge representation primitives and specifies a variety of concrete syntaxes for them. OWL is designed to be compatible with RDF, so that the two together can form an ontology language for the web.

OWL as an Ontology Language for the Semantic Web

- ▷ **Task:** Complement RDF (ABox) with a TBox language.
- ▷ **Idea:** Make use of resources that are values in rdf:type. (called *Classes*)
- ▷ **Definition 12.6.11.** OWL (the *ontology web language*) is a language for encoding TBox information about RDF classes.
- ▷ **Example 12.6.12 (A concept definition for “Mother”).** Mother=Woman \sqcap Parent is represented as

XML Syntax	Functional Syntax
<pre><EquivalentClasses> <Class IRI="Mother"/> <ObjectIntersectionOf> <Class IRI="Woman"/> <Class IRI="Parent"/> </ObjectIntersectionOf> </EquivalentClasses></pre>	<pre>EquivalentClasses(:Mother ObjectIntersectionOf(:Woman :Parent))</pre>

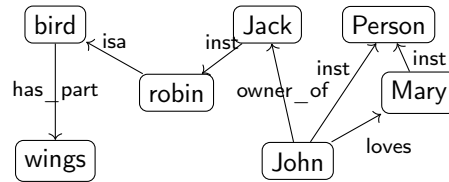
But there are also other syntaxes in regular use. We show the *functional syntax* which is inspired by the *mathematical* notation of relations.

Extended OWL Example in Functional Syntax

- ▷ **Example 12.6.13.** The *semantic network* from Example 12.4.4 can be expressed

in OWL

(in functional syntax)



```

ClassAssertion (:Jack :robin)
ClassAssertion (:John :person)
ClassAssertion (:Mary :person)
ObjectPropertyAssertion (:loves :John :Mary)
ObjectPropertyAssertion (:owner :John :Jack)
SubClassOf (:robin :bird)
SubClassOf (:bird ObjectSomeValuesFrom (:hasPart :wing))
  
```

- ▷ ClassAssertion formalizes the “inst” relation,
- ▷ ObjectPropertyAssertion formalizes [relations](#),
- ▷ SubClassOf formalizes the “isa” relation,
- ▷ for the “has_part” relation, we have to specify that *all birds have a part that is a wing* or equivalently *the class of birds is a subclass of all objects that have some wing*.

We have introduced the ideas behind using description logics as the basis of a “machine-oriented web of data”. While the first OWL specification (2004) had three sublanguages “OWL Lite”, “OWL DL” and “OWL Full”, of which only the middle was based on description logics, with the OWL2 Recommendation from 2009, the foundation in description logics was nearly universally accepted.

The [semantic web](#) hype is by now nearly over, the technology has reached the “plateau of productivity” with many applications being pursued in academia and industry. We will not go into these, but briefly introduce one of the tools that make this work.

SPARQL an RDF Query language

- ▷ **Definition 12.6.14.** [SPARQL](#), the “[SPARQL](#) Protocol and [RDF](#) Query Language” is an [RDF query language](#), able to retrieve and manipulate [data](#) stored in [RDF](#). The [SPARQL](#) language was standardized by the World Wide Web Consortium in 2008 [PS08].
- ▷ [SPARQL](#) is pronounced like the word “*sparkle*”.
- ▷ **Definition 12.6.15.** A system is called a [SPARQL endpoint](#), iff it answers [SPARQL queries](#).
- ▷ **Example 12.6.16.** [Query](#) for person names and their e-mails from a [triplestore](#) with FOAF data.


```

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?email
WHERE {
  ?person a foaf:Person.
  ?person foaf:name ?name.
  
```

```

?person foaf:mbox ?email.
}

```

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 394 2024-02-08

SPARQL end-points can be used to build interesting applications, if fed with the appropriate data. An interesting – and by now paradigmatic – example is the DBpedia project, which builds a large ontology by analyzing Wikipedia fact boxes. These are in a standard [HTML](#) form which can be analyzed e.g. by regular expressions, and their entries are essentially already in triple form: The **subject** is the Wikipedia page they are on, the **predicate** is the key, and the object is either the **URI** on the object value (if it carries a link) or the value itself.

SPARQL Applications: DBpedia

- ▷ **Typical Application:** DBpedia screen-scrapes Wikipedia fact boxes for **RDF** triples and uses **SPARQL** for **querying** the induced **triplestore**.
- ▷ **Example 12.6.17 (DBpedia Query).** People who were born in Erlangen before 1900 (<http://dbpedia.org/snorql>)


```

SELECT ?name ?birth ?death ?person WHERE {
  ?person dbo:birthPlace :Erlangen .
  ?person dbo:birthDate ?birth .
  ?person foaf:name ?name .
  ?person dbo:deathDate ?death .
  FILTER (?birth < "1900-01-01" ^xsd:date) .
}
ORDER BY ?name

```

- ▷ The answers include Emmy Noether and Georg Simon Ohm.

Emmy Noether



Born Amalie Emmy Noether
23 March 1882
Erlangen, Bavaria, German Empire

Died 14 April 1935 (aged 53)
Bryn Mawr, Pennsylvania, United States

Nationality German

Alma mater University of Erlangen

Known for Abstract algebra
Theoretical physics
Noether's theorem

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 395 2024-02-08

A more complex DBpedia Query

- ▷ **Demo:** DBpedia <http://dbpedia.org/snorql/>
- Query:** Soccer players born in a country with more than 10 M inhabitants, who play as goalie in a club that has a stadium with more than 30.000 seats.
- Answer:** computed by DBpedia from a **SPARQL query**

```

SELECT distinct ?soccerplayer ?countryOfBirth ?team ?countryOfTeam ?stadiumcapacity
{
  ?soccerplayer a dbo:SoccerPlayer ;
    dbo:position|dbp:position <http://dbpedia.org/resource/Goalkeeper_(association_football)> ;
    dbo:birthPlace|dbo:country* ?countryOfBirth ;
    #dbo:number 13 ;
    dbo:team ?team .
  ?team dbo:capacity ?stadiumcapacity ; dbo:ground ?countryOfTeam .
  ?countryOfBirth a dbo:Country ; dbo:populationTotal ?population .
  ?countryOfTeam a dbo:Country .
  FILTER (?countryOfTeam != ?countryOfBirth)
  FILTER (?stadiumcapacity > 30000)
  FILTER (?population > 1000000)
} order by ?soccerplayer

```

Results: Browse

SPARQL results:

soccerplayer	countryOfBirth	team	countryOfTeam	stadiumcapacity
:Abdellam_Benabdellah	:Algeria	:Wydad_Casablanca	:Morocco	67000
:Ailton_Moraes_Michellon	:Brazil	:FC_Red_Bull_Salzburg	:Austria	31000
:Aïain_Gouaméné	:Ivory_Coast	:Raja_Casablanca	:Morocco	67000
:Allan_McGregor	:United_Kingdom	:Beşiktaş_J.K.	:Turkey	41903
:Anthony_Scribe	:France	:FC_Dinamo_Tbilisi	:Georgia_(country)	54549
:Brahim_Zaari	:Netherlands	:Raja_Casablanca	:Morocco	67000
:Bréiner_Castillo	:Colombia	:Deportivo_Táchira	:Venezuela	38755
:Carlos_Luis_Morales	:Ecuador	:Club_Atlético_Independiente	:Argentina	48069
:Carlos_Navarro_Montoya	:Colombia	:Club_Atlético_Independiente	:Argentina	48069
:Cristián_Muñoz	:Argentina	:Colo-Colo	:Chile	47000
:Daniel_Ferreira	:Argentina	:FBC_Melgar	:Peru	60000
:David_Bičić	:Czech_Republic	:Karşıyaka_S.K.	:Turkey	51295
:David_Lonia	:Kazakhstan	:Karşıyaka_S.K.	:Turkey	51295
:Denys_Boiko	:Ukraine	:Beşiktaş_J.K.	:Turkey	41903
:Eddie_Gustafsson	:United_States	:FC_Red_Bull_Salzburg	:Austria	31000
:Emilian_Doiha	:Romania	:Lech_Poznań	:Poland	43269
:Eusebio_Acasuzo	:Peru	:Club_Bolívar	:Bolivia	42000
:Faryd_Mondragón	:Colombia	:Real_Zaragoza	:Spain	34596
:Faryd_Mondragón	:Colombia	:Club_Atlético_Independiente	:Argentina	48069
:Federico_Vilar	:Argentina	:Club_Atlas	:Mexico	54500
:Fernando_Martinuzzi	:Argentina	:Real_Garcilaso	:Peru	45000
:Fábio_André_da_Silva	:Portugal	:Servette_FC	:Switzerland	30084
:Gerhard_Tremmel	:Germany	:FC_Red_Bull_Salzburg	:Austria	31000
:Gift_Muzadzi	:United_Kingdom	:Lech_Poznań	:Poland	43269
:Günay_Güvenç	:Germany	:Beşiktaş_J.K.	:Turkey	41903
:Hugo_Marques	:Portugal	:C.D._Primeiro_de_Agosto	:Angola	48500
:Héctor_Landazurí	:Colombia	:La_Paz_F.C.	:Bolivia	42000

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 396 2024-02-08

We conclude our survey of the [semantic web technology stack](#) with the notion of a [triplestore](#), which refers to the [database](#) component, which stores vast collections of [ABox triples](#).

Triple Stores: the Semantic Web Databases

- ▷ **Definition 12.6.18.** A [triplestore](#) or [RDF store](#) is a purpose-built database for the storage [RDF graphs](#) and retrieval of [RDF triples](#) usually through variants of [SPARQL](#).
- ▷ Common [triplestores](#) include
 - ▷ Virtuoso: <https://virtuoso.openlinksw.com/> (used in [DBpedia](#))
 - ▷ GraphDB: <http://graphdb.ontotext.com/> (often used in [WissKI](#))
 - ▷ blazegraph: <https://blazegraph.com/> (open source; used in [WikiData](#))
- ▷ **Definition 12.6.19.** A [description logic reasoner](#) implements of reasoning services based on a satisfiability test for [description logics](#).
- ▷ Common [description logic reasoners](#) include
 - ▷ FACT++: <http://owl.man.ac.uk/factplusplus/>
 - ▷ Hermit: <http://www.hermit-reasoner.com/>
- ▷ **Intuition:** [Triplestores](#) concentrate on [querying](#) very large [ABoxes](#) with partial consideration of the [TBox](#), while [DL reasoners](#) concentrate on the full set of ontology inference services, but fail on large [ABoxes](#).

12.7 Ontologies vs. Databases

To understand [ontologies](#) better and contrast them to [database systems](#) to understand their respective possible role in documenting [cultural artefacts](#). We start off with a definition of the concept and components of an [ontology](#).

We will still keep our presentation of the material at a general level without committing to a particular ontology language or system.

We now consolidate our understanding of all these concepts with an example. We build an ontology by first constructing a [TBox](#) and then a corresponding [ABox](#).

Example: Hogwarts Ontology

- ▷ **Example 12.7.1.** [Axioms](#) describe the structure of the world,

Class HogwartsStudent = Student and attendsSchool Hogwarts
 Class: HogwartsStudent \sqsubseteq hasPet only (Owl or Cat or Toad)
 ObjectProperty: hasPet Inverses: isPetOf
 Class: Phoenix \sqsubseteq isPetOf only Wizard

- ▷ **Example 12.7.2.** [Facts](#) describe some particular concrete situation,

Individual: Hedwig
 Types: Owl
 Individual: HarryPotter
 Types: HogwartsStudent
 Facts: hasPet Hedwig
 Individual: Fawkes
 Types: Phoenix
 Facts: isPetOf Dumbledore

It is very instructive to compare [ontologies](#) to [databases](#). There are some similarities induced by the joint intention to represent structured data, but also some important differences, which will play a crucial role in our discussion later on.

Ontologies vs. Databases

- ▷ **Obvious Analogy:** In an [ontology](#):

- ▷ [axioms](#) analogous to [DB schema](#) (structure and constraints on data)
- ▷ [facts](#) analogous to [DB data](#)
 - ▷ data instantiates schema, is consistent with schema constraints

- ▷ **But there are also important differences:**

Database:	Ontology:
<ul style="list-style-type: none"> ▷ Closed world assumption (CWA) <ul style="list-style-type: none"> ▷ Missing information treated as false ▷ Unique name assumption (UNA) <ul style="list-style-type: none"> ▷ Each individual has a single, unique name ▷ Schema behaves as constraints on structure of data <ul style="list-style-type: none"> ▷ Define legal database states. 	<ul style="list-style-type: none"> ▷ Open world assumption (OWA) <ul style="list-style-type: none"> ▷ Missing information treated as unknown ▷ No UNA <ul style="list-style-type: none"> ▷ Individuals may have more than one name ▷ Ontology axioms behave like implications (inference rules) <ul style="list-style-type: none"> ▷ Entail implicit information

Let us elucidate these quite abstract concepts and differences using a simple example, which we again take from the Hogwarts ontology (see Example 12.7.1 and Example 12.7.2).

DB vs. Ontology by Example (Querying)

▷ **Given the Ontology:**

Individual: HarryPotter

Facts: hasFriend RonWeasley

hasFriend HermioneGranger

hasPet Hedwig

Individual: Draco Malfoy

▷ **Query:** Is Draco Malfoy a friend of HarryPotter?

▷ DB: No

▷ Ontology: Don't Know (OWA: didn't say Draco was not Harry's friend)

▷ **Counting Query:** How many friends does Harry Potter have?

▷ DB: 2

▷ Ontology: at least 1 (No UNA: Ron and Hermione may be 2 names for same person)

▷ **How about:** if we add

DifferentIndividuals: RonWeasley HermioneGranger

▷ DB: 2

▷ Ontology: at least 2 (OWA: Harry may have more friends we didn't mention yet)

▷ **And:** if we also add

Individual: HarryPotter

Types: hasFriend only RonWeasley or HermioneGranger

- ▷ DB: 2
- ▷ Ontology: 2

We continue our example with the behavior if we insert new information to the Hogwarts ontology. Again, [databases](#) and ontology systems react differently.

DB vs. Ontology by Example (Insertion)

- ▷ **Given:** the ontology from Example 12.7.1 and Example 12.7.2 insert

Individual: Dumbledore
 Individual: Fawkes
 Types: Phoenix
 Facts: isPetOf Dumbledore

- ▷ **System Response:**

- ▷ DB: Update rejected: constraint violation
 - ▷ Range of hasPet is Human; Dumbledore is not (CWA)
- ▷ Ontology Reasoner:
 - ▷ Infer that Dumbledore is Human
 - ▷ Also infer that Dumbledore is a Wizard (only a Wizard can have a phoenix as a pet)

Finally, we come to one of the central disciplines in which to compare [databases](#) and ontology based [information systems](#): [query](#) answering. Here we see a crucial difference: ontology [queries](#) are [semantic](#), i.e. they take both [axioms](#) and [facts](#) into account.

DB vs. Ontology by Example: Query Answering

- ▷ DB schema plays no role in [query](#) answering (efficiently implementable)
- ▷ Ontology axioms play a powerful and crucial role in QA
 - ▷ Answer may include implicitly derived facts
 - ▷ Can answer conceptual as well as extensional [queries](#)
E.g., *Can a Muggle have a Phoenix for a pet?*
 - ▷ May have very high worst case [complexity](#) ($\hat{=}$ terrible running time)
[Implementations](#) may still behave well in typical cases.
- ▷ **Definition 12.7.3.** We call a [query language](#) [semantic](#), iff [query](#) answering involves [derived axioms](#) and [facts](#).
- ▷ **Observation 12.7.4.** *Ontology [queries](#) are [semantic](#), while [database queries](#) are not.*

We will now summarize what we have learned about ontology-based information systems.

Summary: Ontology Based Information Systems

- ▷ Analogous to [relational database management systems](#)
Ontology $\hat{=}$ [schema](#); instances $\hat{=}$ [data](#)
- ▷ Some important (dis)advantages
 - + (Relatively) easy to maintain and update schema.
 - ▷ Schema plus data are integrated in a logical theory.
 - + [Query results](#) reflect both schema and [data](#)
 - + Can deal with incomplete information
 - + Able to answer both intensional and extensional [queries](#)
 - Semantics may be counter-intuitive or even inappropriate
 - ▷ Open -vs- closed world; axioms -vs- constraints.
 - [Query](#) answering much more difficult. (based on logical entailment)
 - ▷ Can lead to scalability problems.
- ▷ [In a nutshell](#) they deliver more valuable answers at cost of [efficiency](#).

12.8 Exercises

Problem 8.1 (Function/Argument Form of a Semantic Network)

Write the [semantic network](#) from Example 12.4.10 in [function/argument notation](#).

Problem 8.2 (Evaluation of Semantic Networks)

Using the example from Problem 8.1, discuss the pros and cons – give two of each - of [semantic networks](#).

Problem 8.3 (Semantic Web Technology)

[Semantic web](#) technology comes in two parts, [RDF](#) and [OWL](#). Briefly describe their roles in the [semantic web](#). How do they relate to [ACL](#)?

Problem 8.4

1. [Install](#) the Protege System from <http://protege.stanford.edu/> on your [computer](#) and
2. use it to represent the following [knowledge](#) into an ABox:
 - (a) *Vincent is the brother of Cecilia who is George's daughter.*
 - (b) *Ruth is George's niece and Paul her brother.*
 - (c) *Frida is George's mother.*
3. Define a TBox of family relationships (compliant to the common understanding) that is sufficiently rich so that the following relationships can be inferred (discuss the inferences).
 - (a) *Paul is Cecilia's cousin.*
 - (b) *Frida is Ruth's and Vincent's grandmother.*
 - (c) *George has a brother or sister.*

Chapter 13

The WissKI System: A Virtual Research Environment for Cultural Heritage

We will now come to the [WissKI](#) system itself, which positions itself as a virtual research environment for cultural heritage. Indeed it is a comprehensive, ontology-based information system for documenting, studying, and presenting our [cultural heritage](#).

Before we go into the technicalities of the [WissKI](#) system itself, let us recall the requirements and motivations.

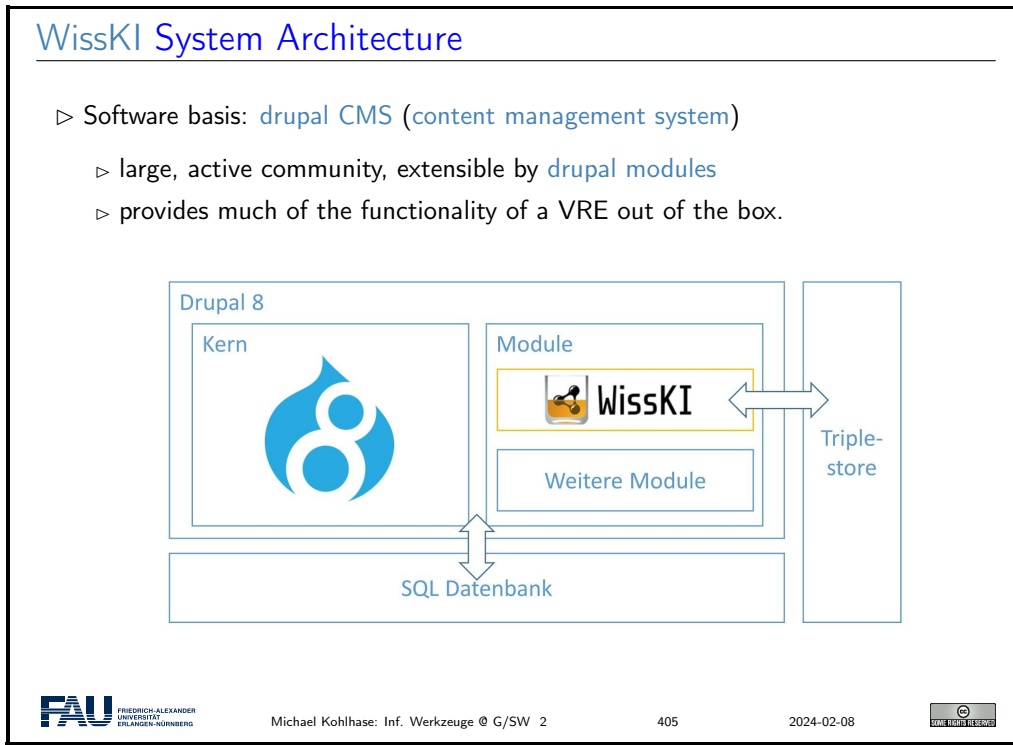
WissKI: a Virtual Research Env. for Cultural Heritage

- ▷ **Definition 13.0.1.** [WissKI](#) is a virtual research environment (VRE) for managing scholarly data and documenting [cultural heritage](#).
- ▷ **Requirements:** For a virtual research environment for [cultural heritage](#), we need
 - ▷ scientific communication about and documentation of the [cultural heritage](#)
 - ▷ networking [knowledge](#) from different disciplines ([transdisciplinarity](#))
 - ▷ high-quality data acquisition and analysis
 - ▷ safeguarding authorship, authenticity, persistence
 - ▷ support of scientific publication
- ▷ [WissKI](#) was developed by the research group of Prof. Günther Görtz at FAU Erlangen-Nürnberg and is now used in hundreds of DH projects across Germany.
- ▷ FAU supports [cultural heritage](#) research by providing hosted [WissKI](#) instances.
 - ▷ See <https://wisski.data.fau.de> for details
 - ▷ We will use an instance for the Kirmes paintings in the homework assignments

13.1 WissKI extends Drupal

The first thing about the [WissKI](#) system is that it is realized as an extension of the [drupal web content management system](#), which already provides many of the features (e.g. user management,

web authoring, collaboration, ...) a VRE needs to [implement](#).



We now give a general overview of the [drupal](#) system, and introduce the concepts we need for understanding [WissKI](#) system. Naturally, this does now do the [drupal WCMS](#) justice. For an introduction we refer readers to [Gla17; Tom17] and the drupal [web site](#) [Dru].

Drupal: A Web Content Management Framework

- ▷ **Definition 13.1.1.** [Drupal](#) is an [open source web content management application](#). It combines [CMS](#) functionality with knowledge management via [RDF](#).
- ▷ **Definition 13.1.2.** [Drupal](#) allows to configure [web pages modularly](#) from content [blocks](#), which can be
 - ▷ [static content](#), i.e. supplied by a [module](#),
 - ▷ [user supplied content](#), or
 - ▷ [views](#), i.e. listings of content fragments from other [blocks](#).

These can be assembled into [web pages](#) via a visual interface: the [config bar](#).

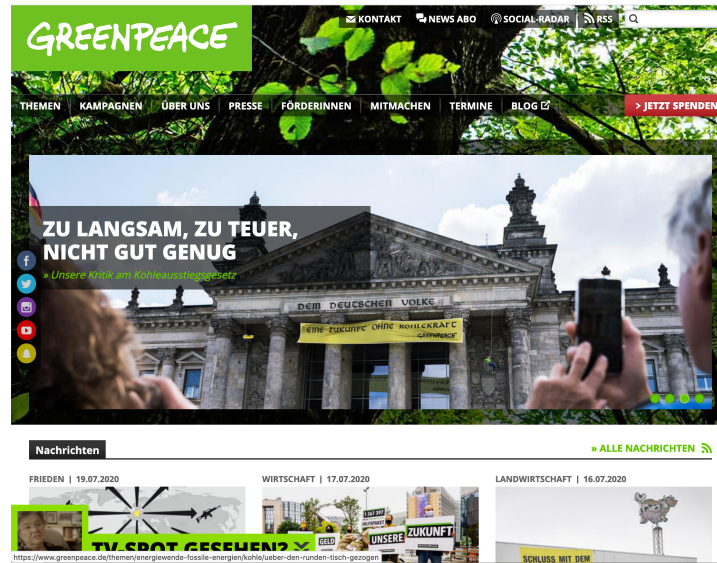
FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 406 2024-02-08

To fortify our intuition about the concepts introduced above, let us try to find them in an existing [web page](#).

Assembling a Web Site via Drupal Blocks (Example)

- ▷ **Example 13.1.3 (Greenpeace via Drupal).** Can you find the blocks?



We now come to one of the most important features used in **WissKI**: **drupal** is **modular** and **extensible**; this allows us to build the features for an ontology-based information system as **drupal modules**.

Drupal Modules and Themes

- ▷ **Idea:** **Drupal** is designed to be **modular** and **extensible** (so it can adapt to the ever-changing web)
- ▷ **Definition 13.1.4 (Modular Design).** **Drupal** functionality is structured into
 - ▷ **drupal core** – the basic **CMS** functionality
 - ▷ **modules** which contribute e.g. new block types (~ 45.000)
 - ▷ **themes** which contribute new UI layouts (~ 2800)

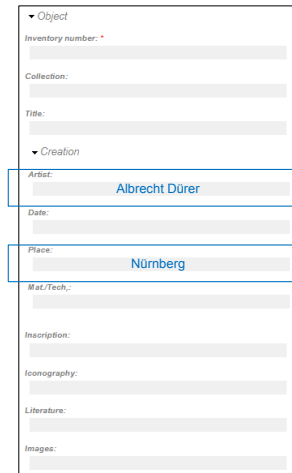
Drupal core is the vanilla system as downloaded, **modules** and **themes** must be **installed** and **configured** separately via the **config bar**.

- ▷ The **drupal core** functionalities include
 - ▷ user/account management
 - ▷ menu management,
 - ▷ RSS feeds,
 - ▷ taxonomy,
 - ▷ page layout customization (via **blocks** and **views**),
 - ▷ system administration

This brings us to the central data acquisition subsystem in [drupal](#), which we will use to build our system. Much of the actual data in the [drupal](#) system is internally stored in terms of [dictionaries](#): systems of [key/value](#) pairs.

Bundles and Fields in Drupal (Data Entry)

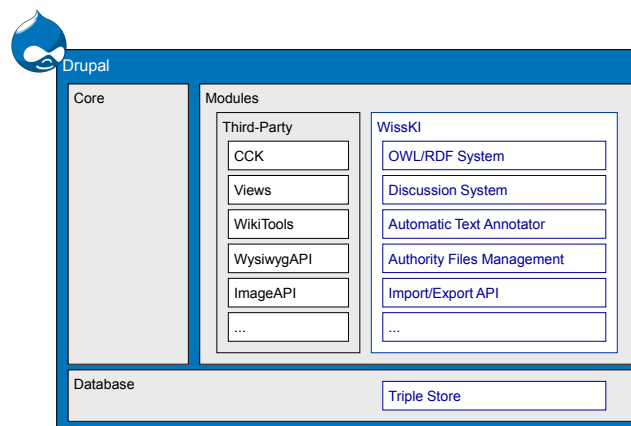
- ▷ **Definition 13.1.5.** [Drupal](#) has a special data type called a [bundle](#), which is essentially a [dictionary](#): it contains [key/value](#) pairs called [fields](#).
 - ▷ [bundles](#) can be nested \rightsquigarrow sub [bundles](#).
 - ▷ [fields](#) also have data type information, etc. to support editing.
- ▷ [drupal](#) presents [bundles](#) as
 - ▷ [HTML](#) lists for reading
 - ▷ [HTML](#) forms for data entry/editing
- ▷ [Drupal bundles](#) induce [blocks](#) that can be used for data entry and presentation.



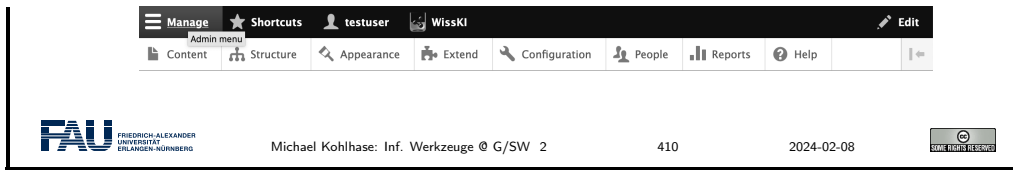
Now we can summarize the [WissKI](#) architecture in a simple equation. While this glosses over many of finer points in the system, it is important to keep this in mind for working with the system in practice.

WissKI System Architecture (Recap)

- ▷ $\text{WissKI} = \text{drupal} + \text{CIDOC CRM} + \text{triplestore} + \text{WissKI modules}$



- ▷ **Note:** Much of [WissKI](#) functionality is configurable via the [drupal config bar](#).



13.2 Dealing with Ontology Paths: The WissKI Pathbuilder

We now come to what is probably the defining feature of **WissKI**: the **WissKI path builder**. It solves the problem that with ontologies, even for simple facts we have to generate entire **ontology paths**.

The WissKI Path Builder (Idea)

- ▷ **Recall:** *Albrecht Dürer painted Melencolia 1 in Nürnberg*

```

graph LR
    m["m : E84"] -- P108i --> q["q : E12"]
    q -- P14 --> d["d : E21"]
    q -- P7 --> p["p : E53"]
    d -- P131 --> a["a : E82"]
    p -- P87 --> n["n : E48"]
    a -- P3 --> d1["A. Dürer"]
    n -- P3 --> d2["Nürnberg"]

```

- ▷ **Idea:** Hide the complexity induced by the ontology from the user
 - ▷ Form-based **interaction** with categories and fields (as in a RDBMS UI)
- ▷ **Definition 13.2.1.** The **WissKI path builder** maps **ontology groups** and **ontology paths** to **drupal bundles** and **fields**.
 - ▷ **ontology groups** become data entry forms (**bundles**) for the root entities,
 - ▷ their **fields** are mapped to **ontology paths**.
 - ▷ subtrees in the ontology become **sub-bundles**. (shared objects)

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2
411
2024-02-08

Even though we have introduced all the necessary concepts above, the best way of understanding this is to look at our running example again: the **path builder** induces a data entry form that allows us to enter a whole set of **ontology paths**, introducing and sharing intermediary objects along the way.

The WissKI Path Builder (Example)

- ▷ **Example 13.2.2 (A WissKI Group).**

FAU
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Michael Kohlhase: Inf. Werkzeuge @ G/SW 2

412

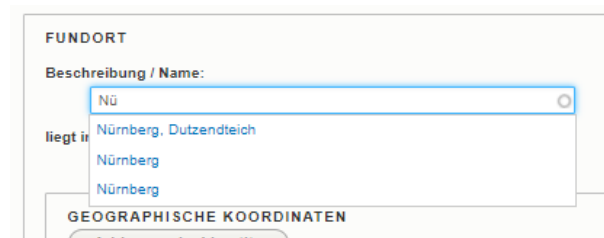
2024-02-08

www.kohlhase.de

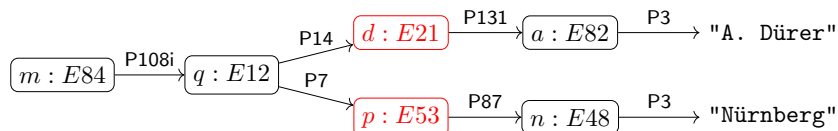
If we look at the data entry form on the left of Example 13.2.2, then we see that we only enter strings, not the objects we mean. So there is the problem of **disambiguating** which objects that are then linked to some object via **CIDOC CRM** relations we actually mean with the string.

Sharing and Disambiguation in Path Builders

- ▷ **Observation 13.2.3.** Sometimes we want to refer to existing entities in *WissKI*.
- ▷ **Example 13.2.4 (Referring to Nürnberg).** (We love tab completion)



- ▷ **Example 13.2.5 (To What).** Albrecht Dürer created all his etchings in Nürnberg.
- ▷ **Problem:** (In paths) we are creating lots of objects, which ones to offer?
- ▷ **Idea:** Mark the entities we might want to reuse on paths while specifying them.
- ▷ **Definition 13.2.6.** A **disambiguation point** in a path marks an entity that can be re used in data acquisition.
- ▷ **Example 13.2.7.** Disambiguation points are highlighted in red on paths.



Now we can have a look at how drupal sees (and shows) path builders

Specifying/Maintaining WissKI Path Builders

- ▷ **Recall:** A WissKI path builder maps ontology groups and ontology paths to drupal bundles and fields.
- ▷ **Example 13.2.8 (Specifying a WissKI Path Builder).**

TITLE	PATH	ENABLED	FIELD TYPE	CARDINALITY	OPERATIONS
Werk	Group [ecrm E22_Man-Made_Object]	<input checked="" type="checkbox"/>		Unlimited	Edit
Titel	ecrm E22_Man-Made_Object -> ecrm P102_has_title -> ecrm E15_Title	<input checked="" type="checkbox"/>	Text (plain)	1	Edit
Verwalter	ecrm E22_Man-Made_Object -> ecrm P50_has_current_keeper -> ecrm E40_Legal_Body -> ecrm P1_is_identified_by -> ecrm E82_Actor_Appellation	<input checked="" type="checkbox"/>	Text (plain)	1	Edit
Inventarnummer	ecrm E22_Man-Made_Object -> ecrm P1_is_identified_by -> ecrm E42_Identifier	<input checked="" type="checkbox"/>	Text (plain)	1	Edit
Beziehung	ecrm E22_Man-Made_Object -> ecrm P46_forms_part_of -> ecrm E22_Man-Made_Object -> ecrm P102_has_title -> ecrm E15_Title	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	Edit
Herstellung	Group [ecrm E22_Man-Made_Object -> ecrm P108_was_produced_by -> ecrm E12_Production]	<input checked="" type="checkbox"/>		Unlimited	Edit
Hersteller	ecrm E22_Man-Made_Object -> ecrm P108_was_produced_by -> ecrm E12_Production -> ecrm P14_carried_out_by -> ecrm E21_Person -> ecrm P131_is_identified_by -> ecrm E82_Actor_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	Edit
Datum	ecrm E22_Man-Made_Object -> ecrm P108_was_produced_by -> ecrm E12_Production -> ecrm P4_has_time-span -> ecrm E52_Time-Span	<input checked="" type="checkbox"/>	Text (plain)	1	Edit
Ort	ecrm E22_Man-Made_Object -> ecrm P108_was_produced_by -> ecrm E12_Production -> ecrm P7_took_place_at -> ecrm E53_Place -> ecrm P1_is_identified_by -> ecrm E44_Place_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	Edit
Material	ecrm E22_Man-Made_Object -> ecrm P108_was_produced_by -> ecrm E12_Production -> ecrm P32_used_general_technique -> ecrm E57_Material -> ecrm P1_is_identified_by -> ecrm E75_Conceptual_Object_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	Edit
Technik	ecrm E22_Man-Made_Object -> ecrm P108_was_produced_by -> ecrm E12_Production -> ecrm P31_used_specific_technique -> ecrm E29_Design_or_Procedure -> ecrm P1_is_identified_by -> ecrm E75_Conceptual_Object_Appellation	<input checked="" type="checkbox"/>	Text (plain)	Unlimited	Edit
Kommentar	ecrm E22_Man-Made_Object -> ecrm P129_is_subject_of -> ecrm E31_Document	<input checked="" type="checkbox"/>	Text (formatted, long)	1	Edit
Abbildung	ecrm E22_Man-Made_Object -> ecrm P138_has_representation -> ecrm E36_Visual_Item -> ecrm P1_is_identified_by -> ecrm E51_Contact_Point	<input checked="" type="checkbox"/>	Image	Unlimited	Edit

FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG Michael Kohlhase: Inf. Werkzeuge @ G/SW 2 414 2024-02-08

Of course all paths of an ontology group can be visualized as a graph. WissKI supports this as well.

WissKI Path Builders as Graphs

- ▷ **Example 13.2.9 (A WissKI Path Construtor as a Graph).**

Very nice and helpful, but does not work currently!

And finally, a **path builder** can be seen as a set of triples indeed this is the default export format for path builders.

Of course all **paths** of an **ontology group** can be visualized as a graph. **WissKI** supports this as well.

WissKI Path Builders as Triples

- ▷ **Of course** we can view **path builders** as sets of triples.
- ▷ **Example 13.2.10 (A WissKI Path Construtor as Triples).**



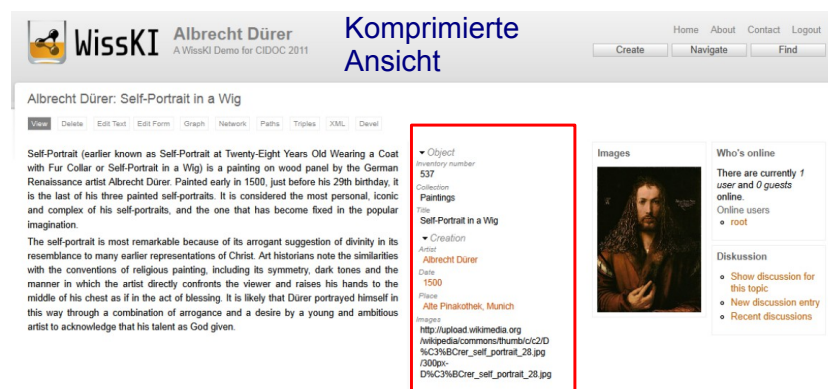
Incoming Subject	Incoming Predicate	Incoming Object
a192abb5-116a-3994-a9b0-05c7acc0da96_text	ecrm:P129_is_about	ecrm:P67_refers_to
a192abb5-116a-3994-a9b0-05c7acc0da96_text		
Outgoing predicate	Outgoing Object	
rdf:type	ecrm:E84_Information_Carrier	
ecrm:P109_was_produced_by	ecrm:E12_Productions6934f80829-1a94-b754-4b40a547b23	
ecrm:P1_is_identified_by	ecrm:E42_Identifier30251719-d60a-5a54-91af-4320a40656	
ecrm:P1_is_identified_by	ecrm:E36_Title6bc34940-8ac9-a774-4424-43020139f6d	
ecrm:P60_has_current_keeper	ecrm:E40_Legal_Bodye633760-0784-0c44-a916-814542a9159	
ecrm:P66_shows_visual_item	ecrm:E38_Imageede530b4-4775-b394-81ca-59c253390270	

- ▷ Such an export also allows standardized communication.

But of course, **path builders** can not only be used as data acquisition devices. They also define **drupal blocks** which can be used for data visualization (akin to fact boxes in Wikipedia).

Data Presentation using Path Builders in WissKI

- ▷ **Path builders** can be used as **drupal blocks** for data presentation.
 - ▷ For every object o , aggregate the values of the paths starting in o .
- ▷ **Example 13.2.11 (Compressed View).**

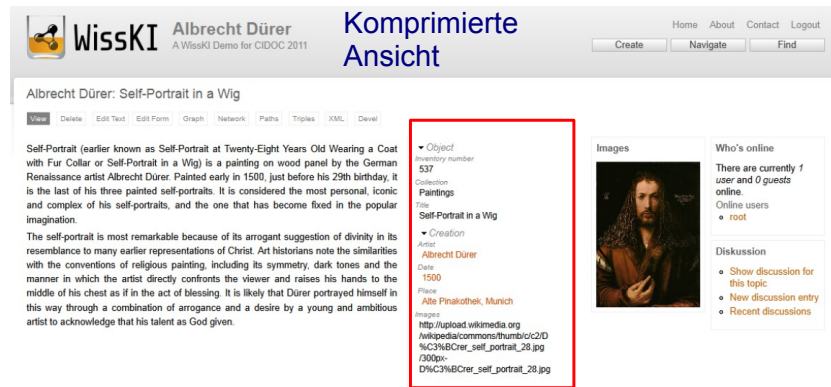


Object
Inventory number: 527
Collection: Paintings
Title: Self-Portrait in a Wig
Creation
Artist: Albrecht Dürer
Date: 1500
Place: Alte Pinakothek, Munich
Image: http://upload.wikimedia.org/wikipedia/commons/thumb/c/c2/NC3%BCrer_self_portrait_28.jpg/300px-DC3%BCrer_self_portrait_28.jpg

13.3 The WissKI Link Block

The WissKI Link Block (Idea)

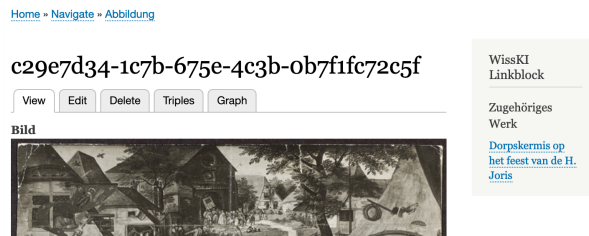
- ▷ **Observation 13.3.1.** For an entity in a *RDF graph*, both the outgoing and the incoming relations are important for understanding.
- ▷ **Example 13.3.2.** This view only shows the outgoing edges!



- ▷ **Idea:** Add a *block* with “incoming links” to the page, use the *path builder*.

Link Blocks (Definition)

- ▷ **Definition 13.3.3.** Let *p* be a *drupal* page for an *ontology group g*, then a *WissKI link block* is a special *drupal block* with associated *path builder*, whose *ontology paths* all end in *g*.
- ▷ **Example 13.3.4 (A link block for Images).**



Note the difference between

- ▷ a “work” – the original painting Pieter Brueghel created in 1628
- ▷ and an “image of the work” – a b/w photograph of the “work”.

This particular **link block** mediates between these two.

A Link Block in the Wild (the full Picture)

▷ **Example 13.3.5 (A link block for Images).**

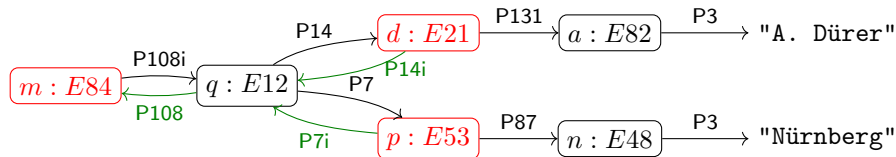
- ▷ **outgoing relations** below the image,
- ▷ **incoming ones** in the link block

Making Link Blocks via the Path Builder

▷ How to make a **link block** in page p for group g ? (Details at [WH])

1. create a **block** via the **config bar** and place it on p .
2. associate it with a **link block path builder**
3. model **paths** into g in the **path builder** (various source groups)

▷ **Idea:** You essentially know **link block** paths already: If you have already modeled a path g, r_1, \dots, r_n, s for a group s , then you have a path $s, r_n^{-1}, \dots, r_1^{-1}, g$, where r_i^{-1} are the inverse roles of r_i (exist in CIDOC CRM)



▷ **Note:** With this setup, you never have to fill out the link block paths!



13.4 Cultural Heritage Research: Querying WissKI Resources

So far, we have concentrated on the [WissKI](#) system, and how that can be used for data acquisition and documentation of [cultural artefacts](#). While we did this we lost view of the most important aspect: what are we doing data acquisition for? Arguably this is [cultural heritage](#) research – and we mean this in an inclusive manner – this could be academic research or researching for a school project or article in a newspaper.

This research and how the [WissKI](#) system can support is what we will go into now.

Research in WissKI

- ▷ **So far** we have seen how to acquire complex [knowledge](#) about [cultural artefacts](#) using [CIDOC CRM ABoxes](#).
- ▷ **Question:** But how do we do research using [WissKI](#)?
- ▷ **Answer:** Finding patterns, inherent connections, . . . in the data.
- ▷ **But how?:** That depends on the kind of research you want to do. Here are some [WissKI](#) research tools
 1. we can use [drupal](#) search on the data.
 2. We can formulate our own [queries](#) in [SPARQL](#)
 3. We can pre-configure various [queries](#) in [drupal views](#).


Michael Kohlhase: Inf. Werkzeuge @ G/SW 2
422
2024-02-08


The simplest form of “research” is just being able to search over the objects that have been created. This is one of the basic facilities [WissKI](#) offers out of the box. Already that can be quite useful.

Drupal Search in WissKI

▷ **Example 13.4.1.**

Search

Search WissKI Entities | Content | Users

Search by Entity Title

Entity Title

Finds titles from the cache table

▼ Advanced Search

in Bundles

Künstler

Abbildung

Werk

in Paths

Künstler

Name (erfassungsmasken.name) contains


Albrecht

Werke dieses Künstlers (pb_wisskilinkblock.werke_dieses_kunstlers) contains

Melencolia

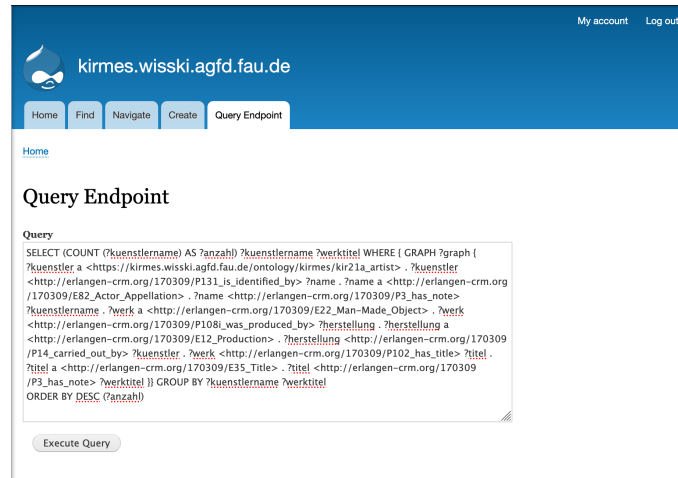
Match

All: Any:


Search Wisski Entities

SPARQL Endpoint in WissKI

- ▷ **Example 13.4.2.** Find kirmes paintings and their painters and count them



My account Log out

kirmes.wisski.agfd.fau.de

Home Find Navigate Create Query Endpoint

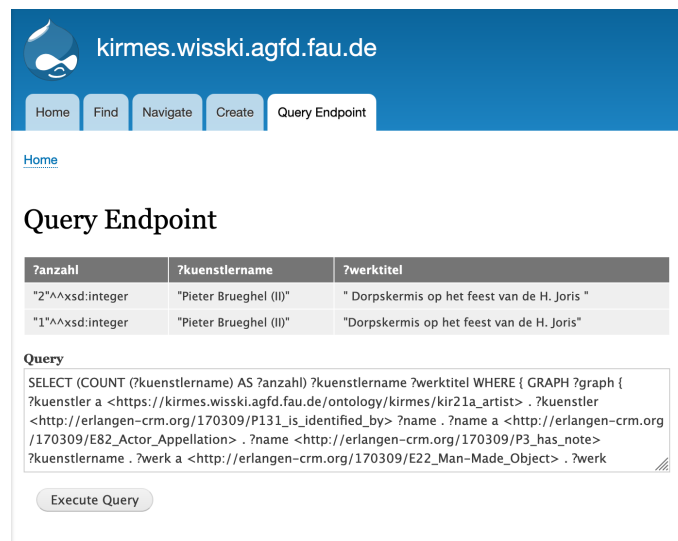
Home

Query Endpoint

Query

```
SELECT (COUNT (?kuenstlername) AS ?anzahl) ?kuenstlername ?werktitel WHERE { GRAPH ?graph {
  ?kuenstler a <https://kirmes.wisski.agfd.fau.de/ontology/kirmes/kir21a_artist> . ?kuenstler
  <http://erlangen-crm.org/170309/P131_is_identified_by> ?name . ?name a <http://erlangen-crm.org/170309/E82_Actor_Appellation> . ?name <http://erlangen-crm.org/170309/P3_has_note>
  ?kuenstlername . ?werk a <http://erlangen-crm.org/170309/E22_Man-Made_Object> . ?werk
  <http://erlangen-crm.org/170309/P1081_was_produced_by> ?herstellung . ?herstellung a
  <http://erlangen-crm.org/170309/E12_Production> . ?herstellung <http://erlangen-crm.org/170309/P14_carried_out_by> ?kuenstler . ?werk <http://erlangen-crm.org/170309/P102_has_title> ?titel .
  ?titel a <http://erlangen-crm.org/170309/E35_Title> . ?titel <http://erlangen-crm.org/170309/P3_has_note> ?werktitel } } GROUP BY ?kuenstlername ?werktitel
ORDER BY DESC (?anzahl)
```

Execute Query



kirmes.wisski.agfd.fau.de

Home Find Navigate Create Query Endpoint

Home

Query Endpoint

?anzahl	?kuenstlername	?werktitel
"2"^^xsd:integer	"Pieter Brueghel (II)"	"Dorpskermis op het feest van de H. Joris"
"1"^^xsd:integer	"Pieter Brueghel (II)"	"Dorpskermis op het feest van de H. Joris"

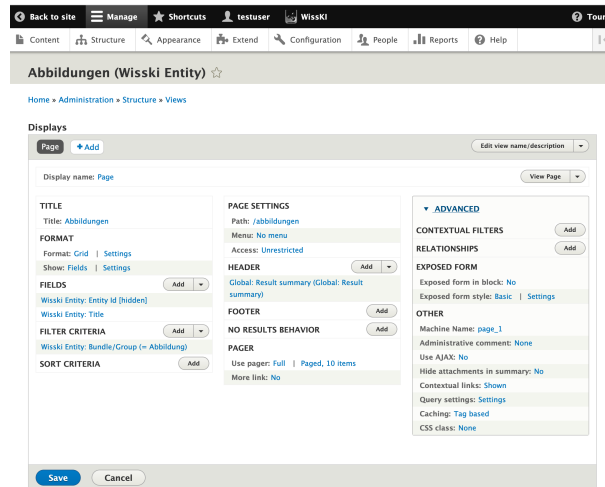
Query

```
SELECT (COUNT (?kuenstlername) AS ?anzahl) ?kuenstlername ?werktitel WHERE { GRAPH ?graph {
  ?kuenstler a <https://kirmes.wisski.agfd.fau.de/ontology/kirmes/kir21a_artist> . ?kuenstler
  <http://erlangen-crm.org/170309/P131_is_identified_by> ?name . ?name a <http://erlangen-crm.org/170309/E82_Actor_Appellation> . ?name <http://erlangen-crm.org/170309/P3_has_note>
  ?kuenstlername . ?werk a <http://erlangen-crm.org/170309/E22_Man-Made_Object> . ?werk
```

Execute Query

Data Presentation via Views in WissKI

- ▷ **Example 13.4.3 (Configuring a View).** This makes a [drupal block](#).



Drupal generates a **SPARQL query**, aggregates **results** into a **block**.

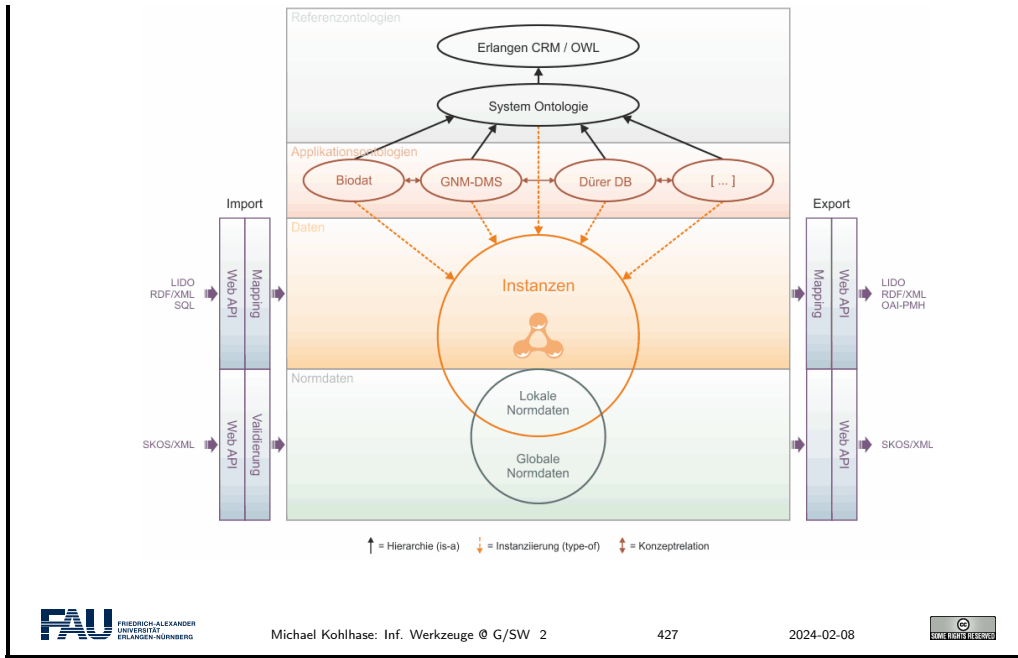
This Research is WissKI-instance-local

- ▷ **Observation 13.4.4.** *All these research queries only work in the current **WissKI** instance.*
 - ▷ **Observation 13.4.5.** *There is probably much more about the entities you are interested in outside your particular **WissKI** instance.*
 - ▷ **Problem:** How to make use of this?
 - ▷ **Solution:** We need to do two things
 1. Make use of other people's ABoxes
 2. Provide your ABox to other people.
- This practice is called **linked open data**. (up next)

13.5 Application Ontologies in WissKI

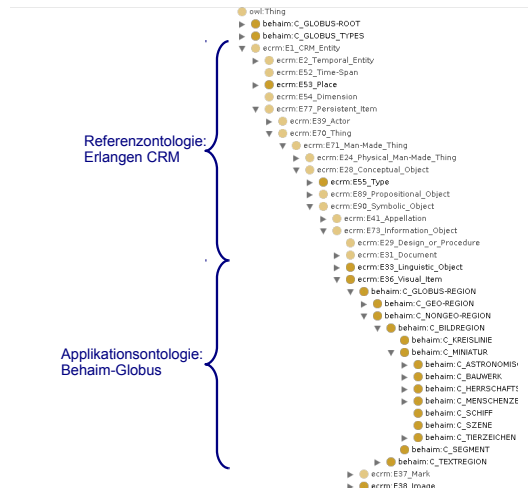
WissKI Information Architecture (Ontologies)

- ▷ Ontologies, instances, and export formats



Application Ontologies extend CIDOC CRM

- ▷ **Observation 13.5.1.** Sometimes we need more than *CIDOC CRM*.
- ▷ **Definition 13.5.2.** A *WissKI application ontology* is one that extends *CIDOC CRM*, without changing it.
- ▷ **Example 13.5.3 (Behaim Application Ontology).**



Making an Application Ontology

- ▷ The “current ontology” of a [WissKI](#) instance can be configured via the [config bar](#) via the “WissKI ontology” [module](#).
- ▷ The [application ontology](#) should import [CIDOC CRM](#).
- ▷ **Idea:** Use [Protg](#) for that.

13.6 The Linked Open Data Cloud

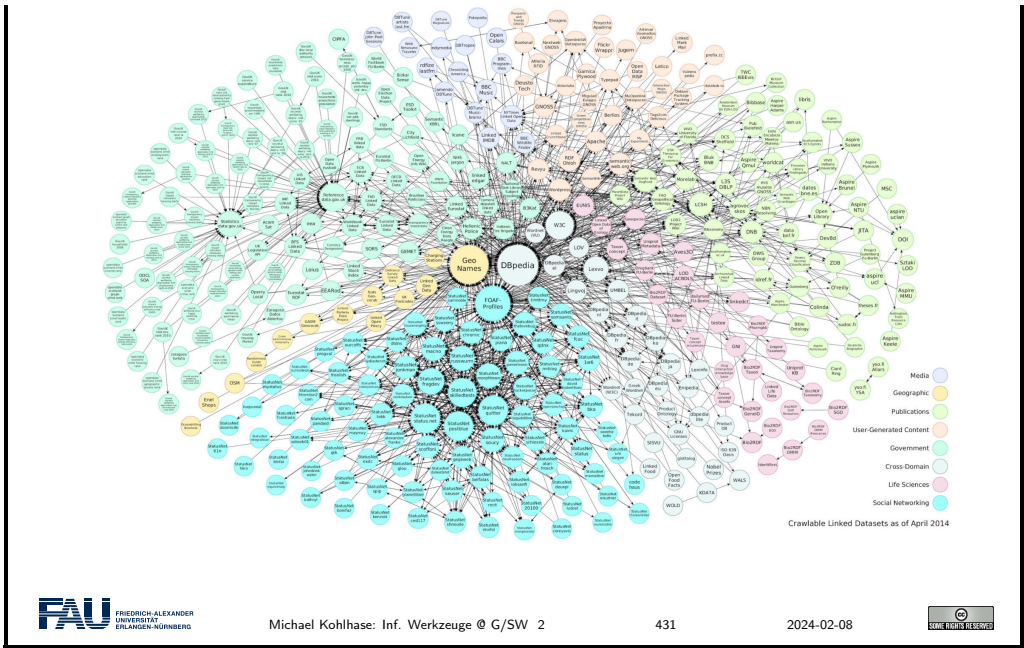
Linked Open Data

- ▷ **Definition 13.6.1.** [Linked data](#) is structured data in which classified objects are interlinked via [relations](#) with other objects so that the data becomes more useful through [semantic queries](#) and access methods.
- ▷ **Definition 13.6.2.** [Linked open data \(LOD\)](#) is [linked data](#) which is released under an [open license](#), which does not impede its reuse by the community.
- ▷ **Definition 13.6.3.** Given the [semantic web](#) technology stack, we can create interoperable ontologies and interlinked data sets, we call their totality the [LOD](#).
- ▷ **Recall the LOD Incentives:**
 - ▷ incentivize other authors to extend/improve the LOD
 - ↪ more/better data can be generated at a lower cost.
 - ▷ generate *attention* to the LOD and recognition for authors
 - ↪ this gives alternative revenue models for authors.

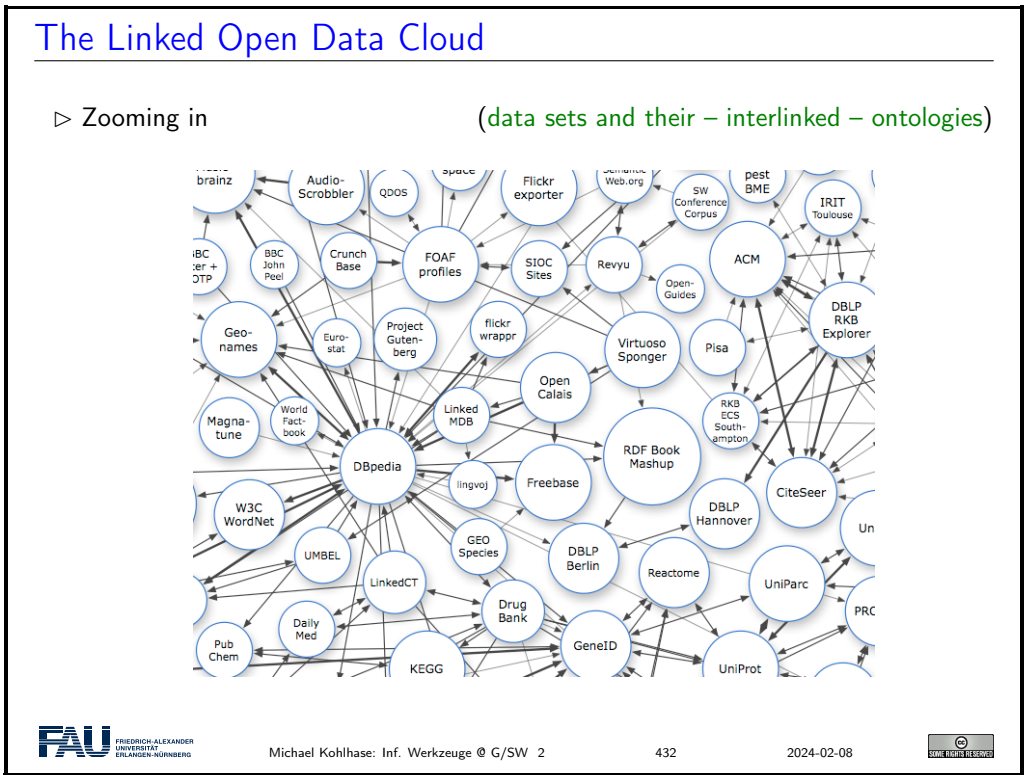
By Definition 13.6.3 the [linked open data cloud](#) is the totality of [linked open data](#) that has been published. [LOD] tracks (the larger parts of) it. This gives us a sense of the extent of this giant network of [knowledge](#) expressed as triples.

The Linked Open Data Cloud

- ▷ The [linked open data cloud](#) in 2014 (today much bigger, but unreadable)



We now “zoom in” on this picture to get a better sense”. Each of the circles in the picture is a data set of at least 1000 triples. The DBpedia in the center of this fragment has 3 billion triples alone (in 2014).



The ideas of the [linked open data cloud](#) directly apply [knowledge](#) about [cultural artefacts](#) as we formalize them in the [WissKI](#) system: we can directly reference objects from the cloud in [WissKI](#).

Using the LOD-Cloud in WissKI

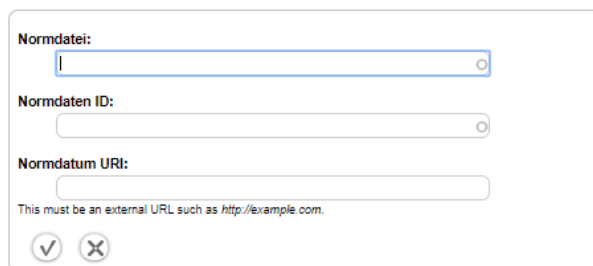
- ▷ **Idea:** Do not re-model entities that already exist (in the LOD Cloud)
- ▷ **Problem:** Most of the LOD Cloud is about things we do not want.
- ▷ But there are some sources that are useful
 - ▷ the **GND (Gemeinsame Normdatei [GND])**, an authority file for personal/corporate names and keywords from literary catalogs,
 - ▷ **geonames[GN]**, a geographical **database** with more than 25M names and locations
 - ▷ Wikipedia
- ▷ **Observation 13.6.4.** All of them provide **URIs** for real world entities, which is just what we need for objects in **RDF triples**.
- ▷ **Definition 13.6.5.** **WissKI** provides special **modules** called **adapters** for **GND** and **geonames**.

Using **linked open data** in **WissKI** actually makes for higher-quality digitizations, as they are more interoperable. Unfortunately, **WissKI** only supports the two adapters we mention above. There are many many more that would be useful.

Let us now see how to concretely use an adapter, here for the geonames service.

Using Geonames in WissKI (Example)

1. **Example 13.6.6.** We want to use the “Meilwald” (Erlangen) in **WissKI**.
2. make a sub-ontology groups “norm data” in the **WissKI path builder**
3. The induced sub-bundle looks like this:



The screenshot shows a form with three input fields:

- Normdatei:** An empty text input field.
- Normdaten ID:** An empty text input field.
- Normdatum URI:** An empty text input field.

Below the fields, there is a note: "This must be an external URL such as <http://example.com>." At the bottom of the form, there are two circular buttons: a checkmark (✓) and a cross (✗).

4. We enter `https://geodata.org` for “Normdatei” and go there to find out the **URI** for “Meilwald” which goes into “Normdatum **URI**”.



The GeoNames geographical database covers all countries and contains over eleven million placenames that are available for download free of charge.

Meilwald all countries [\[advanced search\]](#)

enter a location name, ex: "Paris", "Mount Everest", "New York"

5. there may be multiple results

(here only one)

Name	Country	Feature class	Latitude	Longitude
1 Erlanger Meilwald Erlanger Meil-Wald,Erlanger Meilwald,Meilwald	Germany, Bavaria	forest(s)	N 49° 36' 30"	E 11° 1' 39"

6. Select/click the intended one, check the details

7. Enter the URL from the URL bar into "Normdatum URI".

Normdatei:

Normdaten ID:

Normdatum URI:

This must be an external URL such as <http://example.com>.

If we – as we did here – tell the story of using authority files in *WissKI* from a *linked open data* perspective, a curious asymmetry becomes apparent: *WissKI* is using *LOD* resources, but is – by and large – not contributing *LOD* resources back to the “public domain” of *linked open cultural*

heritage data.

Towards a [WissKI Commons](#) in the LOD Cloud

- ▷ **Recap:** We can directly refer to (URLs of) external objects in [WissKI](#).
- ▷ **Observation 13.6.7.** *The most interesting source for references to [cultural artefacts](#) are other [WissKI](#) instances.*
- ▷ **Problem:** A [WissKI](#) is an island, unless it exports its data! (few do)
- ▷ **Idea:** We need a LOD cloud of [cultural heritage research data](#) under to foster object centric research in the humanities.
- ▷ **Definition 13.6.8.** We call the part of this resource that can be created by aggregating [WissKI](#) exports the [WissKI commons](#).
- ▷ **Observation 13.6.9.** *[WissKI](#) exports meet the [FAIR](#) principles quite nicely already.*
- ▷ We will be working on a FAU [WissKI commons](#) in the next years. (help wanted)

This asymmetry is a very serious problem, since [cultural heritage](#) research is not profiting as much from digitizations as it could. Keeping data in [WissKI](#) silos – this is what we do when we are not exporting [WissKI](#) data and referencing objects from other [WissKI](#) instances – leads to fragmentation of the research community and to duplication of work.

Bibliography

- [BHK16] Jens Bove, Lutz Heusinger, and Angela Kailus. *Marburger Informations-, Dokumentations- und Administrations-System (MIDAS): Handbuch und CD*. 4th ed. K.G.Saur, 2016. DOI: 10.11588/artdok.00003770.
- [CC] *CIDOC CRM - The CIDOC Conceptual Reference Model*. URL: <http://www.cidoc-crm.org/> (visited on 07/13/2020).
- [CQ69] Allan M. Collins and M. Ross Quillian. “Retrieval time from semantic memory”. In: *Journal of verbal learning and verbal behavior* 8.2 (1969), pp. 240–247. DOI: 10.1016/S0022-5371(69)80069-1.
- [DCM12] DCMI Usage Board. *DCMI Metadata Terms*. DCMI Recommendation. Dublin Core Metadata Initiative, June 14, 2012. URL: <http://dublincore.org/documents/2012/06/14/dcmi-terms/>.
- [Dru] *Drupal.org – Community plumbing*. URL: <http://drupal.org> (visited on 02/14/2015).
- [ECRMa] *erlangen-crm*. URL: <https://github.com/erlangen-crm> (visited on 07/13/2020).
- [ECRMb] *Erlangen CRM/OWL - An OWL DL 1.0 implementation of the CIDOC Conceptual Reference Model (CIDOC CRM)*. URL: <http://erlangen-crm.org/> (visited on 07/13/2020).
- [FAIR18] European Commission Expert Group on FAIR Data. *Turning FAIR into reality*. 2018. DOI: 10.2777/1524.
- [FOAF14] *FOAF Vocabulary Specification 0.99*. Namespace Document. The FOAF Project, Jan. 14, 2014. URL: <http://xmlns.com/foaf/spec/>.
- [Gla17] Matt Glaman. *Drupal 8 Development Cookbook – Harness the power of Drupal 8 with this recipe-based practical guide*. 2nd ed. Packt Publishing, 2-17. ISBN: 9781788290401.
- [GN] *Geonames*. URL: <https://www.geonames.org/> (visited on 07/29/2020).
- [GND] *DNB – The Integrated Authority File (GND)*. URL: https://www.dnb.de/EN/Professionell/Standardisierung/GND/gnd_node.html (visited on 07/29/2020).
- [Her+13a] Ivan Herman et al. *RDF 1.1 Primer (Second Edition). Rich Structured Data Markup for Web Documents*. W3C Working Group Note. World Wide Web Consortium (W3C), 2013. URL: <http://www.w3.org/TR/rdfa-primer>.
- [Her+13b] Ivan Herman et al. *RDFa 1.1 Primer – Second Edition. Rich Structured Data Markup for Web Documents*. W3C Working Group Note. World Wide Web Consortium (W3C), Apr. 19, 2013. URL: <http://www.w3.org/TR/xhtml1-rdfa-primer/>.
- [HiDa] *HiDa*. URL: <https://www.startext.de/produkte/hida> (visited on 07/12/2020).
- [Hit+12] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer (Second Edition)*. W3C Recommendation. World Wide Web Consortium (W3C), 2012. URL: <http://www.w3.org/TR/owl-primer>.
- [JS] *json – JSON encoder and decoder*. URL: <https://docs.python.org/3/library/json.html> (visited on 04/16/2021).

- [KC04] Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation. World Wide Web Consortium (W3C), Feb. 10, 2004. URL: <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
- [LM] *LabelMe: the open annotation tool*. URL: <http://labelme.csail.mit.edu> (visited on 08/28/2020).
- [LOD] *The Linked Open Data Cloud*. URL: <https://lod-cloud.net/> (visited on 08/19/2020).
- [LXML] *lxml – XML and HTML with Python*. URL: <https://lxml.de> (visited on 12/09/2019).
- [Nor+18a] Emily Nordmann et al. *Lecture capture: Practical recommendations for students and lecturers*. 2018. URL: <https://osf.io/huydx/download>.
- [Nor+18b] Emily Nordmann et al. *Vorlesungsaufzeichnungen nutzen: Eine Anleitung für Studierende*. 2018. URL: <https://osf.io/e6r7a/download>.
- [OWL09] OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation. World Wide Web Consortium (W3C), Oct. 27, 2009. URL: <http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>.
- [PMDA] *Python – MySQL Database Access*. URL: https://www.tutorialspoint.com/python/python_database_access.htm (visited on 11/18/2018).
- [Pro] *Protégé*. Project Home page at <http://protege.stanford.edu>. URL: <http://protege.stanford.edu>.
- [PRR97] G. Probst, St. Raub, and Kai Romhardt. *Wissen managen*. 4 (2003). Gabler Verlag, 1997.
- [PS08] Eric Prud’hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. World Wide Web Consortium (W3C), Jan. 15, 2008. URL: <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
- [SR14] Guus Schreiber and Yves Raimond. *RDF 1.1 Primer*. W3C Working Group Note. World Wide Web Consortium (W3C), 2014. URL: <http://www.w3.org/TR/rdf-primer>.
- [SSU04] Susan Schreibman, Ray Siemens, and John Unsworth, eds. *A Companion to Digital Humanities*. Wiley-Blackwell, 2004. ISBN: 978-1-405-10321-3. URL: <http://www.digitalhumanities.org/companion>.
- [SUMO] *Suggested Upper Merged Ontology*. URL: <http://www.adampease.org/OP/> (visited on 01/25/2019).
- [Tom17] Todd Tomlinson. *Enterprise Drupal 8 Development – For Advanced Projects and Large Development Teams*. Apress, 2017. ISBN: 9781484202548.
- [Tur95] Sherry Turkle. *Life on the Screen: Identity in the Age of the Internet*. Simon & Schuster, 1995.
- [UL] *urllib – URL handling modules*. URL: <https://docs.python.org/3/library/urllib.html> (visited on 04/15/2021).
- [WH] *WissKI Handbuch*. URL: http://wiss-ki.eu/documentation/wisski_handbuch (visited on 07/23/2020).
- [Wil+16] Mark D. Wilkinson et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific Data* 3 (2016). DOI: 10.1038/sdata.2016.18.