

# Database-driven Web Technology (LIX021B05)

Instructor: Çağrı Çöltekin

`c.coltekin@rug.nl`

Information science/Informatiekunde

November 11, 2013

# Who, where, when

Course	Database-driven Web Technology (LIX021B05) 2013/14
Instructor	Çağrı Çöltekin
Email	<a href="mailto:c.coltekin@rug.nl">c.coltekin@rug.nl</a>
Lectures	Mon 11:00–13:00, 1313.0338
Labs	Thu 11:00–13:00, 1312.0107A
Office hours	Thu 09:00–11:00, H1311.0426 (or by appointment)
Course page	<a href="http://www.let.rug.nl/coltekin/courses/dbweb2013/">http://www.let.rug.nl/coltekin/courses/dbweb2013/</a>

# Literature

There is no compulsory textbook for this course.

- ▶ A good database book, for example one of the following, will be handy to have at hand.
  - ▶ *Database System Concepts* by A. Silberschatz, H. F. Korth & S. Sudarshan.
  - ▶ *Database Management Systems* by Ramakrishnan & Gehrke
  - ▶ *A First Course in Database Systems* by Ullman & Widom
  - ▶ *Fundamentals of Database Systems* by Elmasri & Navathe
  - ▶ *Database Systems: The Complete Book* by Garcia-Molina, Ullman & Widom
  - ▶ *Database Systems: A Practical Approach to Design, Implementation, and Management* by Connolly & Begg
- ▶ You will be referred to online sources if/when necessary.

## What you should already know

**Databases** You are assumed know basic relational database design concepts and SQL. We will only have a quick review/reminder session.

**HTML** You are assumed know the basics of HTML. We will only revisit/review HTML forms in this course.

**Programming** You are assumed to have some experience with programming, and some earlier exposition to PHP (from the web programming course). If not, you should be able to pick it up quickly.

## After this course . . .

You should be able to

- ▶ develop interactive server-side web applications with a relational database back end
- ▶ develop a relational database for a real-world application
- ▶ understand what happens behind the browser in a web-based application
- ▶ be able to identify potential security problems in web-based applications
- ▶ be familiar with performance issues for large scale web-based applications using databases.

# Time plan

Week	Lecture (Wed)
1	Introduction & organization
2	Meeting with teams: no course
3	Web programming & Accessing databases from PHP
4	Session management
5	Security
6	Summary & QA & Discussion
7	Project (prototype) demonstrations

# Evaluation

- ▶ Project: 70%.
  - ▶ Requirements and initial design report (Due date: Nov 26).
  - ▶ Project (prototype) demonstration (Dec 17).
  - ▶ Finalized project (Jan 31).
- ▶ Individual homeworks: 30%.
  - ▶ Homeworks will consist of steps of a small similar project.

You need a minimum of 5.5 (from each component) to pass.

## About projects

Project form the main part of this course.

- ▶ Ideal team size is 3,  $\pm 1$  is also fine.
- ▶ Contribution by all members are compulsory.
- ▶ Naturally, you will share tasks, however, everybody should understand the project they are working on fully.
- ▶ You are required to use a version management system for all project related files (more on this today).
- ▶ There is a single project subject. But, your end results will differ based on your requirement collection as well as your choices to implement additional features.

Time is short: you need to act quickly to team up, and start working.



## Project subject

Current topic (tentative): A web-based crowd-sourcing application.

- ▶ All teams will be working on the same topic.
- ▶ The project will be finalized by Friday, and announced on Nestor.
- ▶ You may affect the project subject: if you have an idea it is not time to talk to me.
- ▶ You will collect the full requirements yourself by interviewing the user(s). (next week)

# Projects and deadlines

				1	2	3	
	4	5	6	7	8	9	10
November 2013	11	12	13	14	15	16	17
	18	19	20	21	22	23	24
	25	26	27	28	29	30	1
December 2013	2	3	4	5	6	7	8
	9	10	11	12	13	14	15
	16	17	18	19	20	21	22
	23	24	25	26	27	28	29
	30	31	1	2	3	4	5
January 2014	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
	20	21	22	23	24	25	26
	27	28	29	30	31		

- 22 Team formation and requirements collection.
- 02 Requirements documentation & initial project report.
- 06 Prototype demonstration & presentation.
- 31 Deadline for the finalized project. **No extensions!**

## Project evaluation

- ▶ Good database design
- ▶ Secure and efficient web programming
- ▶ Continuous effort

There are two early steps that that contribute to the evaluation:

- ▶ An initial project report (about 3 pages, due 2013-12-02). It should include,
  - ▶ a summary of the project requirements
  - ▶ your Initial database design (e.g., E-R design, DB schema)
  - ▶ type(s) of users, typical queries expected, security (authentication, authorization) requirements.
- ▶ Project demonstration/presentation (2014-01-06): you will present an early version/prototype, explain planned work for the final version.

# Rest of today

Today:

- ▶ A quick refresher on PHP & HTML form processing.
- ▶ A quick introduction to software version control using git.

# HTML forms

```
<form action="form.php" method="post" >  
Name: <input type="text" name="name" >  
Password: <input type="password" name="pass" >  
</form >
```

- ▶ HTML forms are defined in `<form>` tag.
- ▶ **action** attribute specifies the URL that will handle the form input when submitted.
- ▶ **method** is either **GET** or **POST**.
- ▶ Input elements are specified inside the form.
- ▶ `<input>` tag specifies most common input types, but some elements have different tags, e.g., `<textarea>`.
- ▶ You can specify a default value, typically using **value** attribute.

# Input types

**text** A single line text field

**password** Same as text, but the input is not displayed

**textarea** A multi-line text field

**radio** Radio button, among a set of buttons user can select only one

**checkbox** Like radio buttons, but user can select multiple (or none)

**option/select** Create drop-down lists.

**button** A button

**img** A button with an image

**file** A file upload button

**hidden** A static value.

## GET or POST ?

### ▶ GET method,

- ▶ Encodes the form output in the URL:

`http://my.hostname/form.php?name=myname&age=myage`

+ users can bookmark it.

- the values are visible on the URL bar (passwords?).
- there may be size limitation. No set standard, but rule of thumb: no more than about 2K characters.

### ▶ POST method,

- ▶ Encodes the form output in the message body.

```
POST /form.php HTTP/1.1
```

```
header: value
```

```
....
```

```
name=myname&age=myage
```

- + with the help of HTTP, you can pass values securely.
- + it can handle much more data.
- you cannot bookmark the output of a form submission.

# PHP and forms

- ▶ Form output from PHP is easy, it is just HTML output.
- ▶ If a form uses your PHP script as **action**, you can access the form input using two super globals,
  - ▶ `$_POST` if you used `method=post`
  - ▶ `$_GET` if you used `method=get`
- ▶ PHP also provides a unified associative array `$_REQUEST`, which combines both `$_POST` and `$_GET`.
- ▶ All these variables are associative arrays. For example, if you had a form input with `name="username"`, the value could be accessed using `$_REQUEST['username']`.



## A simple form example

```

1 <!-- This is a simple HTML form
2 -->
3 <form action=form.php method=get>
4 Username: <input type="text"
5           name="name" /><br>
6 Password: <input type="password"
7           name="pass" /><br>
8           <input type="submit"
9           name="submit"
10          value="submit" >
11 </form>

```

```

1 <!-- this is a PHP script that
2     processes it -->
3
4 <?php
5
6 if (!isset($_REQUEST['submit'])) {
7     echo "Why are you here?";
8 } else {
9     echo "Your name is: "
10        . "${_REQUEST['name']}<br>";
11    echo "And password: "
12        . "${_REQUEST['pass']}<br>";
13 }
14 ?>

```

# Is this familiar?

Frist day after the project is assigned ...

```
mak@company $ mkdir proj
mak@company $ ls proj
index.html
```

After a week ... !!!!!

```
mak@company $ ls proj
header.php          header1.php      header2.php
header_current.php index.html       index.html.bkp
index.html.old
```

After a fortnight .....  
.....

```
mak@company $ ls proj
archive           footer.php          footer.php.latest
footer_final.php  header.php          header1.php
header2.php       header_current.php GodHelp
index.html        index.html.bkp     index.html.old
messed up        main_index.html    main_header.php
never used       new_footer.php     new
old              old_data           todo
TODO.latest     toShowManager     version1
version2        webHelp
:
:
:
```

From <http://maktoons.blogspot.nl/2009/06/if-dont-use-version-control-system.html>

# Version Control Systems

A **version control system** (or, *revision control* or *source control* system) is an indispensable tool in software development.

A VCS,

- ▶ Records a history of all changes to all files under VC.
- ▶ Allows going back in time: you can go back to any past state recorded in VCS.
- ▶ Allows inspecting which change happened when.
- ▶ Allows maintaining multiple **branches** of the same software without multiple copies.
- ▶ Allows sandboxing: you can try (experimental) changes without disrupting the 'working copy'.
- ▶ Facilitates team work.
- ▶ It can also be used for other purposes, for example, web pages, documents . . .

## VCS: some terms

**repository** is the database where the VCS stores the code, versions and associated information.

**working copy** is where the programmer makes the changes.

**tag** is a name given to the state of the repository at a certain time.

**branch** is a *virtual* copy of the whole repository for a specific purpose.

**check-in** operation updates the repository using the working copy.

**merge** brings code in different branches (possibly from different programmers) together.

**conflict** may occur during a merge, if same segment of the code was changed in different ways.

## Some VC systems

**RCS** Dates back to early 1980's. Typically single user, single system. Keeps track of single files.

**CVS** Client-server system with support for multiple users on a network. Server side manages the central repository, client side keeps the users' working copies.

**SVN** An improvement over CVS.

**GNU arch, mercurial, bazaar, git . . .** are distributed VCSs. Every working copy keeps a repository. The repository for each programmer is local, but can be merged using different methods.

## Git: introduction

- ▶ Git is a distributed VCS.
- ▶ Developed (primarily) for Linux kernel, but used by many projects.
- ▶ Available for almost all operating systems.
- ▶ Can be shared through HTTP, SSH, git, rsync, email ...
- ▶ Many (somewhat) free software-hosting providers for projects using git. Just a few: GitHub, Gitorious, Bitbucket ...

## Git: starting a new repository

### git init

- ▶ Initializes an empty git repository in the current directory (you should already be in your 'project directory').
- ▶ All repository files live in `.git` subdirectory.
- ▶ You rarely touch files under `.git` (although, you can if you know what you are doing).

## Git: starting a new repository

### git init

- ▶ Initializes an empty git repository in the current directory (you should already be in your 'project directory').
- ▶ All repository files live in `.git` subdirectory.
- ▶ You rarely touch files under `.git` (although, you can if you know what you are doing).

```
mkdir testproject
```

```
cd testproject
```

```
git init
```

```
$EDIT index.php #=====>
```

```
<html>
```

```
<head></head>
```

```
<body>
```

```
<?php
```

```
    echo "Hello Wrold!\n";
```

```
?>
```

```
</body>
```

```
</html>
```



## Git: recording changes in the repository

### `git add` and `git commit`

- ▶ Git has a two-stage commit process. First you need to **add** the changes you want to commit. (This adds the change to a 'cache', called 'index', but we are not concerned with that.)
- ▶ Then, **commit** updates the repository with the changes.

## Git: recording changes in the repository

### `git add` and `git commit`

- ▶ Git has a two-stage commit process. First you need to **add** the changes you want to commit. (This adds the change to a 'cache', called 'index', but we are not concerned with that.)
- ▶ Then, **commit** updates the repository with the changes.

```
git add index.php
```

```
git commit -m "added index.php"
```

## Git: recording changes in the repository

### `git add` and `git commit`

- ▶ Git has a two-stage commit process. First you need to **add** the changes you want to commit. (This adds the change to a 'cache', called 'index', but we are not concerned with that.)
- ▶ Then, **commit** updates the repository with the changes.

```
git add index.php
```

```
git commit -m "added index.php"
```

- ▶ `-m` option to **commit** gives a short comment. Otherwise, git fires up a text editor to write a comment.
- ▶ **Comments are important.** Do not skip and be descriptive.

## Git: inspecting changes

```
git diff
```

# Git: inspecting changes

```
git diff
```

```
# found a bug!  
# edit index.php, change 'Wrold' to 'World'  
git diff  
---- a/index.php  
+++ b/index.php  
@@ -4,7 +4,7 @@  
  <?php  
- echo    "Hello Wrold!";  
+ echo    "Hello World!";  
  ?>
```

## Git: inspecting changes

```
git diff
```

```
# found a bug!  
# edit index.php, change 'Wrold' to 'World'  
git diff  
---- a/index.php  
+++ b/index.php  
@@ -4,7 +4,7 @@  
 <?php  
- echo    "Hello Wrold!";  
+ echo    "Hello World!";  
?>
```

- ▶ Without options, `diff` will give you differences between the working copy and the `HEAD` of the repository for all files.
- ▶ You can inspect differences for any set of files between any two states of the repository using `diff` (more on this later).

# Git: checking the status

## git status

```
git status
# On branch master
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified:      index.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

# Git: checking the status

## git status

```
git status
# On branch master
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified:   index.php
#
no changes added to commit (use "git add" and/or "git commit -a")
```

- ▶ **status** gives you a summary of the current state of your working copy.
- ▶ Let's commit our bug-fix:

```
git commit -a -m "fixed a typo"
```



# Git: logs

## git log

```
git log
```

```
commit 933f1de6b727d7816df9420aa4f0c85c00455d19
```

```
Author: Cagri Coltekin <c.coltekin@rug.nl>
```

```
Date: Sun Nov 6 15:59:31 2011 +0100
```

```
    fixed a typo
```

```
commit 7742fd6ff2a90372bf9545732a9e6923588d3052
```

```
Author: Cagri Coltekin <c.coltekin@rug.nl>
```

```
Date: Sun Nov 6 15:31:56 2011 +0100
```

```
    added inex.php
```

# Git: working with branches

## git branch and git checkout

- ▶ Branches are fast and cheap: use them!
- ▶ Branches can be used for maintaining long-term different versions of software, or short term experimental changes.

```

$ git branch experimental # we are about to do a major change create
$ git checkout experimental # a new branch and switch to it
Switched to branch "experimental"
$ git status # let's see if everything is as expected
# On branch experimental
nothing to commit (working directory clean)
$ $EDITOR index.php # change 'Hello' to 'Hi'
$ $EDITOR LICENSE # also add a LICENSE file
$ git status # check the status again
# On branch experimental
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
# modified:   index.php
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
# LICENSE
no changes added to commit (use "git add" and/or "git commit -a")

```

## Git: working with branches (2)

```
$ git add LICENSE # add the new file
$ git add index.php # add the changes to index.php
$ git status # checking once more doesn't hurt
# On branch experimental
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: LICENSE
# modified: index.php
#
$ git commit -m "Major change in index.php + added LICENSE"
$ ls
index.php LICENSE
$ grep World index.php
    echo "Hi World!";
$ git checkout master
$ ls
index.php
$ grep World index.php
    echo "Hello World!";
```

- ▶ Note: your commits should better be atomic.

## Git: merging branches

### git merge

```
$ git merge experimental
Updating 933f1de..e07aa49
Fast forward
 index.php | 2    +-
 1 files changed, 1 insertions(+), 1 deletions(-)
 create mode 100644 LICENSE
```

- ▶ this merge succeeds with no conflicts
- ▶ now both branches are identical. We can delete the `experimental` branch if we like with the command `git branch -d experimental`.
- ▶ if there were conflicts, we'd need to resolve them manually (helpful tools exist).

## Git: interacting with other repositories

### `git clone`, `git pull` and `git push`

- ▶ `clone` makes a complete copy of a repository.
- ▶ As well as local files, a repository can be accessed (and cloned) through a number of ways. Including, `ssh`, `http`.
- ▶ This allows git to be used in a client-server-like setup.
- ▶ Once you cloned an original source, you can use `git pull` to receive updates, and `git push` to push your changes.
- ▶ You can `pull` from or `push` to multiple repositories.
- ▶ A number of software hosting sites are primarily dedicated for git.

## Git: graphical user interfaces

`git gui`, `gitk`, `qgit` ...

- ▶ the native gui `git gui` allows you to do most of these operations by point&click.
- ▶ a few others exist: `gitk`, `qgit`, [Git Extensions](#) (for Windows), [GitX](#) (for Mac) ... See <http://git-scm.com/tools> for more.
- ▶ Hosting sites, e.g., GitHub, BitBucket, also provide some web-based functionality.

## Git (or other VCS) in your projects

- ▶ We did not even cover the tip of the iceberg, git (and most other VCSes) provide many possibilities that makes your life easier in the long run.
- ▶ More documentation and pointers to good tutorials can be found at <http://git-scm.com/documentation>.
- ▶ Take it seriously in your projects.
- ▶ The time you invest to learn to use a VCS will pay off.
- ▶ You are required to point me to a repository that I can inspect for your projects.

## Summary & next week

Today:

- ▶ Some refresher on PHP & HTML forms processing.
- ▶ A quick introduction to software version control using git.

Next week:

- ▶ Form your team, and arrange a project meeting within next week.
- ▶ Homework part 1.
- ▶ No course.