Data Persistence (for web applications)

Jakub Daněk (Yoso Czech s.r.o.)

Parameter passing, JDBC, Transactions

https://danekja.org (GitHub & LinkedIn links)
http://yoso.fi (yep, in Finnish)

Motivation

- **Many web applications manage persistent data**
- **Persistence implementation impacts many non-functional requirements**
  - Performance
  - Security
  - Durability

Main Topics

- WHERE to store (RDBMS, NoSQL DBMS, File system, other service)
- HOW to work with the data (querying, parameter passing)
- CONSISTENCY - transactions
- Connection management

Storage Options

RDBMS

MariaDB, MySQL, PostgreSQL, Oracle SQL, Microsoft SQL Server, Derby, H2

- **Great for storing relational data – tables and relations**
- **Mature, well tested projects**
  - Minimum bugs
  - Good vendor and community support
  - Good tools

Storage Options

NoSQL

- **Large family of database engines with specific uses**

Document-based: MongoDB, CouchDB

- Good for storing data which dont have fixed structure/tree data
- Limited support for expressing relations
  - Good when individual records do not need to keep any reference to each other
  - Bad when you need relational consistency

Storage Options

NoSQL

- **Large family of database engines with specific uses**

Key-Value: Redis

- Optimized for storing key-value pairs (where value can be a structured object or array)

- Very fast: great for caches, indexes

Storage Options

NoSQL

- **Large family of database engines with specific uses**

Wide-Column: Cassandra, DynamoDB

- Organize data by columns instead of rows
- Great scalability and partitioning
- Good in cases where
  - Writes exceed reads a lot
  - Very little updates and deletes
  - e.g. log storage, tracking information storage

Storage Options

NoSQL

- **Large family of database engines with specific uses**

Graph: Neo4J, Giraph

- **Optimized for storing graph data and querying over them**
  - Maps, ontologies (used for machine reasoning)

Storage Options

NoSQL

- **Shared (mostly) features:**
  - Scalability and partitioning over consistency
    - Most are eventually consistent (sooner or later they get into consistent state, but no promises on when)
    - Poor transaction support
    - Good replication support (it is easier to get done when you don't need to do ACID)

Storage Options

NoSQL

- **Summary**

  - NoSQL databases are great for the tasks they were made for → **not a replacement** of relational databases, more of a **complement**!

  - Always consider your use-cases and requirements before you introduce another piece of technology into your system

Storage Options

File System

- **Easy to use, each application has it out of the box, but…**
  - When scaling horizontally, nodes do not have access file system of others without a lot of work (→ we lose transparency)
  - Scaling filesystem is hard (both space- and performance-wise)
  - Limited indexing, caching, querying support…
- **→ simple to use, but not very suitable for production cases**

Storage Options

Other options

- Many applications for specific cases with focus on querying, caching, scalability...
- Alfresco – file storage with access management, scaling, metadata and query support
- Elastic Search – storage engine with focus on data search and analytics, great for indexing log or big data
- Amazon S3 – object storage with WS API, good for storing data which are not accessed very often (backups etc)
- Object databases – e.g. ObjectDB, storage of objects instead of mapping them into relational model

Storage Options – Further reading

https://db-engines.com/en/

http://nosql-database.org/

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

Working with Data Stores

When working with data storage, there are several points of interest:

- Data Model
- Security
- Performance

Data Modelling and Organization

- **Data structure and organization within persistent storage impacts:**
  - Performance
    - Quite common case: fast read and fast write are contradicting requirements
  - Difficulty of working with the persistence
    - How many requests to the database I need to do to get what I need?
    - Can I easily influence what data I get (full records vs only subset of data)
  - Consistency, space requirements

Data Modelling and Organization

Example: Table normalization for relational databases

Normalized model:

- **Each piece of information is stored only once**
  - → less space needed
  - → updates are fast
- **Reading full data often requires JOINs/multiple selects**
  - → worse performance
  - RDBMS optimize the queries, does not disrupt standard data management applications

Data Modelling and Organization

Example: Table normalization for relational databases

Model designed for write-once, read and analyze afterwards:

- **Single Table which contains all required data for (a set of) query(-ies)**
  - → precomputed data, often duplicated across tables
  - → updates are slow, quite often it is difficult to maintain consistency
  - → reads are fast – over one table, often precomputed values
- **This model is often used in data warehouses**
  - Data storages designed for data analysis, not for transactional processing (your standard data management application)
- **But you should know this from KIV/DB2 ;)**

Data Access Security

- **Two security aspects to consider when working with persistent storage:**

- **Can someone intercept communication with the storage**
  - e.g. when storage server (database, any cloud storage such as GDrive or Dropbox) is a remote to you application server
  - Standard rules for communication between servers apply (we will talk about this in the web-security lecture)

- **Can someone trick my application into executing unwanted queries/writes?**

Data Access Security

- **Can someone trick my application into executing unwanted queries/writes?**

- **Every piece of user input is potentially dangerous when you just append it to the query string (so-called Injection attack):**

  - SQL: “SELECT * FROM user where username = ‘” + username + “’;”
    - What if user provides something like “= any’ OR 1 = 1; <execute arbitrary query here>”
  - FILE: listFiles(“/myapplication/data/” + dirName);
    - What if we have “/myapplication/conf” with passwords and user inputs “../conf” for dirName?

Data Access Security

- **How to prevent injection attacks?**
  - Always validate input parameters
    - SQL connection libraries across languages do that for you
      - JDBC: query = conn.prepareStatement("SELECT * from User where username = ?");
        query.setString(1, username)
    - When working with system file storage:
      - Set proper access rights
      - Ensure in code that your file path has not exited the data directory
      - If possible, deny use of path traversing strings (../, etc.)
    - Any storage you use – investigate how prevent users from accessing data they are not supposed to

Data Access Performance

Two main aspects:

- How long the actual query processing takes

- How many queries do I send to the data store

Data Access Performance

How long the actual query processing takes

- **Usually shows when there is more than little data.**
- **Relational databases suffer from this issue more**
  - Mainly because they allow complicated queries
  - NoSQL databases usually process simple queries → faster
- **Depends mainly on your data model strucutre**

Data Access Performance

How long the actual query processing takes

- **Relational databases suffer from this issue more**
  - Provide indexes for your most common/expensive queries
  - Good indexes prevent searching through all table rows
  - Bad/overused indexes decrease performance though (especially any write queries)

Data Access Performance

How many queries do I send to the data store

- **Shows bad data access implementation**
- **SELECT N+1 problem**
  - 1 query to fetch list of N items
  - N queries to fetch additional data for each of them

Data Access Summary

- **We try to pretend in our application logic layer that we don't care what data store we use**
  - SW developers would like to believe they can leave that to database experts
  - But you need to be aware of the store type and its properties in order to ensure good performance and correct behaviour
  - No way around it, sorry :)

Data Consistency Management

- **Quite often you need to perform multiple actions on a datastore (or several) during processing a single request**
- **These actions should often all succeed or all fail**
  - If one action/write query fails all previous changes need to be rollbacked
- **→ we need Transactions**

Transactions

- **Transaction is a unit of work which either fully completes or fully fails – no partial changes are allowed**
  - If transaction fails, the outside world shouldn't experience any changes
- **ACID**
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Transactions

- **ACID**
  - Atomicity
    - Transaction is an atomic operation – either all happens or nothing
  - Consistency
    - Changes within a transaction do not break any integrity constraints
  - Isolation
    - Concurrent transactions do not influence each other – to a transaction T1 it appears that all other transactions occurred either before or after T1
  - Durability
    - After transaction commits (succeeds), the changes are permanent and survive system failure

Src: https://en.wikipedia.org/wiki/Transaction_processing

Transactions - Isolation

- **To allow concurrency, it is possible to lower isolation of transactions**

- **Concurrency needs are defined by the following events:**
  - Dirty Reads
  - Nonrepeatable Reads
  - Phantoms

Transactions - Isolation

- **Dirty Reads**
  - When T1 reads data written by T2 before T2 is committed
- **Nonrepeatable Reads**
  - T1 reads data
  - Then T2 commits update/delete to those data
  - T1 repeats the same query and gets the updated data (different column values, some rows can be deleted)

Transactions - Isolation

- **Phantoms**
  - T1 reads data query
  - T2 commits insert or update which modifies attribute(s) which formed the search condition
  - T1 executes the same query and gets different results (rows)

Transactions – Isolation (Locks)

- **Sidestep: we need this to understand the isolation levels**
- **There are two types of record locking**
  - READ locks – multiple transactions can read the record, but transaction which wants to WRITE has to wait until all READ locks are released
  - WRITE locks – record is locked for other transaction no matter whether they want to read or write

Transactions – Isolation levels

Isolation levels defined by SQL standard based on which of the described phenomena may occure:

Transaction isolation level	Dirty reads	Nonrepeatable reads	Phantoms
Read uncommitted	X	X	X
Read committed	--	X	X
Repeatable read	--	--	X
Serializable	--	--	--

Src: https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels

Transactions – Isolation levels

- **Read uncommitted**
  - Transactions are not isolated from each other – best concurrency at the cost of consistency
  - Most often used for read-only transactions (to limit harm done to other transactions)
- **Read committed**
  - Only committed data are read
  - Transaction holds read lock while it works with the record
  - Transaction holds write lock until it commits or rollbacks

Transactions – Isolation levels

- **Repeatable read**
  - Transaction holds all READ and WRITE locks of records it returns until commit or rollback → no other transactions can update the rows

- **Serializable**
  - Transaction locks all rows which fit into it's query range, e.g.:
    - "SELECT * FROM Orders" locks the entire Orders table
    - "DELETE FROM Orders WHERE Status = 'CLOSED'" denies any change of rows with status CLOSED, insert or deletion of any rows with status CLOSED
    - That is the difference to the previous level which locks only rows it actually touches (they have to exist and fit query criteria at the time of query)

Transactions – Isolation levels

- **Most often READ_COMMITTED is good compromise between isolation and concurrency**
- **Example where it is insufficient: Dynamic ID generation based on previous record**
  - ID Format: <username><day><id>
    - Where ID is 3-digit number of records created for the given person on the given day
    - e.g. danekja20181031001 (first record created by danekja on 2018-10-31)

Transactions – Isolation levels

Example where it is insufficient: Dynamic ID generation based on previous record

- Algorithm:

```
String lastId = dao.getLastId("danekja", today());
int lastNb = parseID(lastId);
int nextNb = lastNb++;
String nextId = buildId("danekja", today(), nextNb);

dao.createNewRecord(nextId, recordData);
```

- All these steps must happen in complete isolation

Transactions – Isolation levels

Example where it is insufficient: Dynamic ID generation based on previous record

- All these steps must happen in complete isolation

- If another thread attempts to create new record it must wait for the previous transaction to commit (otherwise it gets the same lastId and both transactions will attempt to insert record with same primary key → the later will fail on unique constraint

  - Solution A: set isolation level for the method creating new record on Serializable.
  - Solution B: set isolation level to Repeatable Read and when querying for the lastId, lock the row with WRITE lock (see SQL: SELECT … FOR UPDATE)

  - Solution B prevents lot of locking comparing to A but may be insufficient under certain circumstances

Transactions

- **Initially only common inside DB engines**
- **Current (web) systems require transactions to span across multiple applications, servers**
  - Distributed transactions – see Java Transaction API or OpenXA
  - High-level transaction management which allows management and coordination of transaction on individual resource (e.g. database) level
    - Remember when you see @Transactional annotation in our labs

Transactions

- **Implementation of distributed transaction systems is beyond scope of this course, we will focus on practical use**
- **Ideally all your update application logic methods/functions should be run in single transaction**
  - Another reason why you don't want to have application logic in web layer – you can configure transactions on application layer for all possible front-ends
- **We will show during ORM lecture that some implementations require also read operations to be run in a transaction**

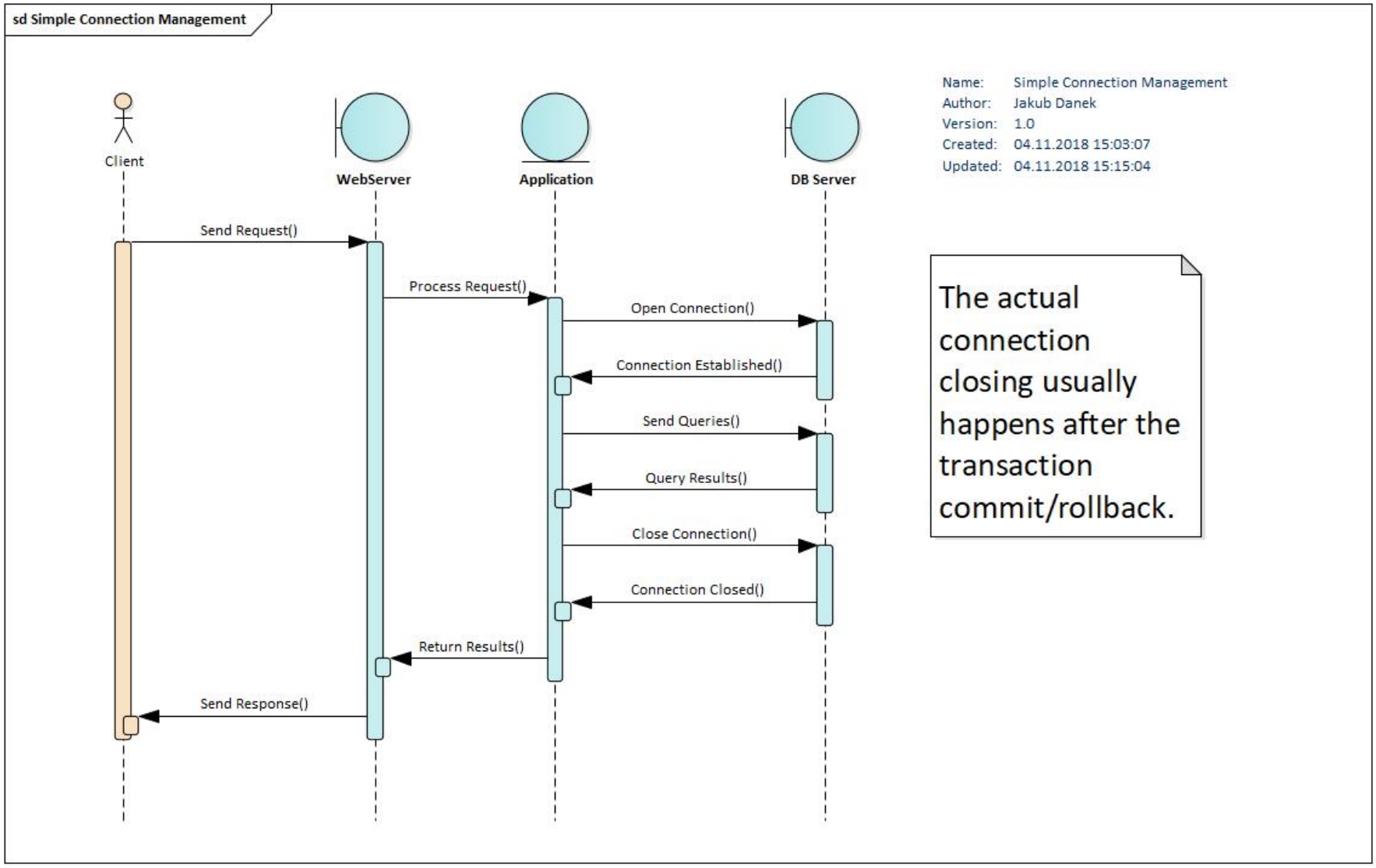
Transactions – Additional Reading

https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/transaction-isolation-levels?view=sql-server-2017

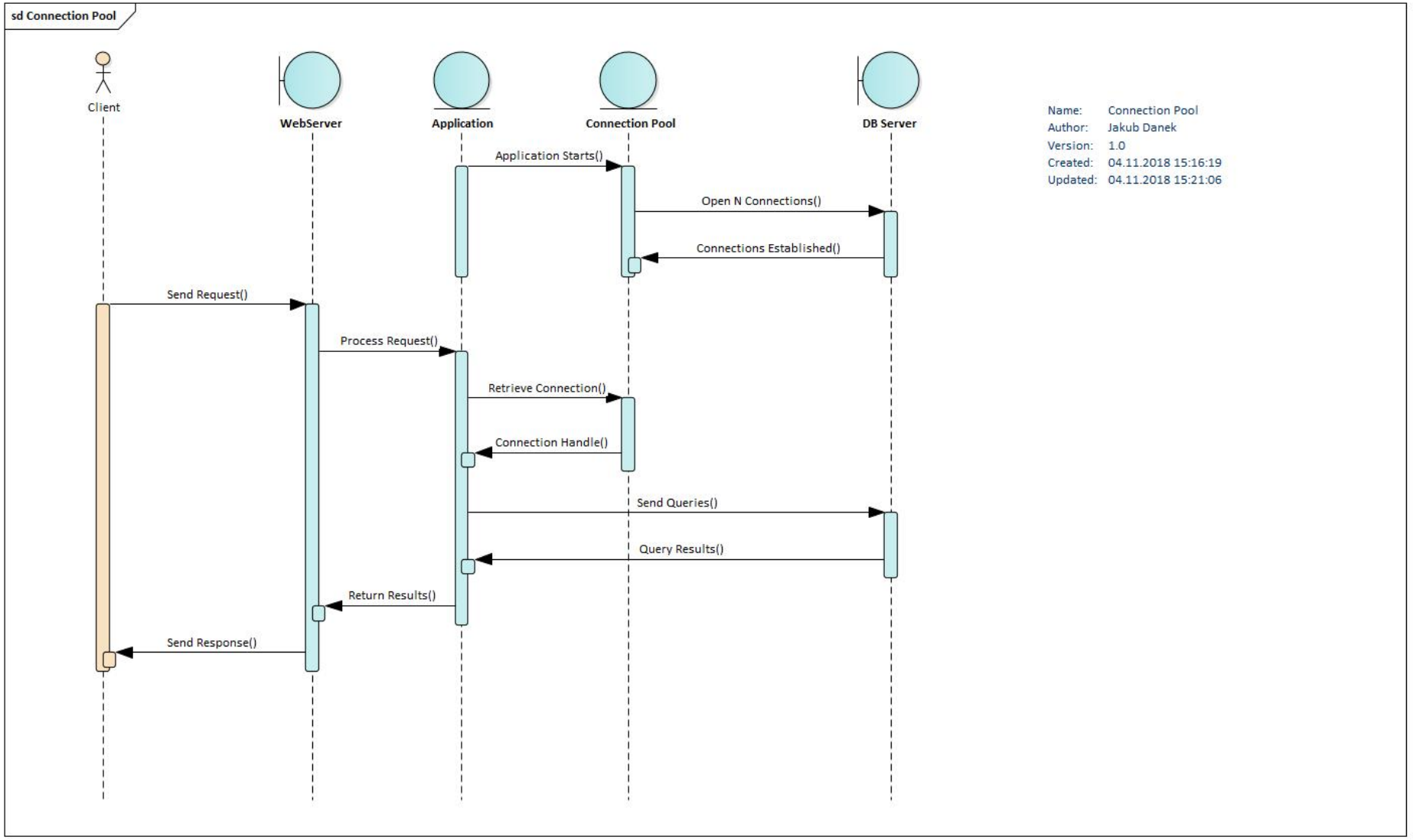
https://en.wikipedia.org/wiki/Transaction_processing

https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cpn2/index.html

Connection Management (for Databases)



Connection Management (for Databases)



Connection Pool

- **Prevents opening new connection between application and db server for each request**
  - → reduced overhead related to the connection negotiation
- **Suitable in cases where connection attributes do not change with each request**
  - e.g. when each user has own credentials for DB access, you cannot use connection pool

Connection Pool

- **Standard properties:**
  - Minimum connections: the pool always has this amount of connections open
  - Maximum connections: how many connections the pool may open
  - Timeout: idle time after which connections above minimum are released
  - Check query: which query the pool uses to check if connection is still valid
  - And other...
- **Most of the properties influence stability/performance of the connection pool**

Connection Pool – Additional Reading

Python:

https://pynative.com/python-database-connection-pooling-with-mysql/

Java:

https://www.mchange.com/projects/c3p0/

.NET:

https://docs.microsoft.com/cs-cz/dotnet/framework/data/adonet/connection-pooling

Node.js:

https://github.com/mysqljs/mysql#pooling-connections

Ruby on Rails:

https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/ConnectionPool.html