Designing and Building Scalable Web Applications

Lecture 1 / 24.10.2022

Agenda

- Course structure and practicalities
- Learning objectives
- The Big Picture
- Defining scalability
- Brief recap of CS-C3170 Web Software Development
- Measuring web application performance
- First course project

Structure and Practicalities

Structure and Practicalities

- Weekly lectures on Mondays from 14:15 to 16:00 (online)
- Weekly readings
  - Typically two to three articles based on which you will create multiple-choice questions (MCQs)
  - Answering and rating multiple choice questions by others
- Three projects (self-, peer-, and teacher-reviewed)
- Course platform (for creating and answering questions and for projects) at https://fitech101.aalto.fi/designing-and-building-scalable-web-applications/
  Note! When you register on the platform, use your @aalto.fi -email address

Grading

- Weekly readings and multiple choice questions: up to 3600 points
  - 150 points per a *good quality* multiple choice question → up to 450 points per week
  - 10 points per answered multiple choice question → up to 150 points per week
- Three projects
  - Possibility to fail, pass, and complete with merits
  - Self-, peer-, and teacher-reviewed
- Grading:
  - Grade 5: At least 3000 points, completed all three projects, at least two of them with merits
  - Grade 4: At least 2700 points, completed all three projects, at least one of them with merits
  - Grade 3: At least 2400 points, completed all three projects, none with merits
  - Grade 2: At least 2100 points, completed two projects, at least one with merits
  - Grade 1: At least 1800 points, completed two projects

NB! Completing a project includes also completing project reviews.

Grading

Checked during final grading - points given by the platform can be removed at that point.

- Weekly readings and multiple choice questions: up to 3600 points
  - 150 points per a *good quality* multiple choice question → up to 450 points per week
  - 10 points per answered multiple choice question → up to 150 points per week
- Three projects
  - Possibility to fail, pass, and complete with merits
  - Self-, peer-, and teacher-reviewed
- Grading:
  - Grade 5: At least 3000 points, completed all three projects, at least two of them with merits
  - Grade 4: At least 2700 points, completed all three projects, at least one of them with merits
  - Grade 3: At least 2400 points, completed all three projects, none with merits
  - Grade 2: At least 2100 points, completed two projects, at least one with merits
  - Grade 1: At least 1800 points, completed two projects

NB! Completing a project includes also completing project reviews.

Schedule

Week / release	Lecture	Readings and MCQs due	Answering and rating MCQs due	Projects due	Project reviews due
(1)	Mon 24.10. 14-16	Fri 28.10. 23:59	Mon 31.10. 23:59		
(2)	Mon 31.10. 14-16	Fri 4.11. 23:59	Mon 7.11. 23:59	Project 1: Sun 6.11. 23:59	Project 1: Wed 9.11. 23:59
(3)	Mon 7.11. 14-16	Fri 11.11. 23:59	Mon 14.11. 23:59		
(4)	Mon 14.11. 14-16	Fri 18.11. 23:59	Mon 21.11. 23:59		
(5)	Mon 21.11. 14-16	Fri 25.11. 23:59	Mon 28.11. 23:59	Project 2: Sun 27.11. 23:59	Project 2: Wed 30.11. 23:59
(6)	Mon 28.11. 14-16	Fri 2.12. 23:59	Mon 5.12. 23:59		
(7)	-			Project 3: Sun 11.12. 23:59	Project 3: Wed 14.12. 23:59

It is possible to return course work 14 days after the deadline and still have the course work included in grading. For MCQs, returning and answering them late reduces given points to the half. For course projects, returning them late means that receiving merits is not possible.

Multiple Choice Questions?

What is *good quality* in multiple choice questions?

- Requires reading the relevant content and thinking about the answers.
- Does not verbatim copy of the content.
- Already the question has meaning: “Which of the following options are true” (not good) vs. “What are the characteristics of the Deno serve function?” (better)
- All answer options should be plausible: “The serve function starts a web server” (plausible, correct), “The serve function defines what to do with incoming requests” (plausible, incorrect), “The serve function returns a Response object for each Request” (plausible, incorrect), “It mimics a dinosaur” (not plausible, incorrect).
- Asks, e.g., about knowledge, understanding, application, analysis, synthesis, and evaluation of contents (getting to know where these come from, read about the Bloom’s taxonomy)

Learning and multiple choice questions?

- Self-explanation effect
- Generation effect
- Testing effect

Learning and multiple choice questions?

- Self-explanation effect

In general, learners who explain content to themselves learn better than those who do not

- Generation effect
- Testing effect

Learning and multiple choice questions?

- Self-explanation effect
- Generation effect
- Testing effect

In general, learners who explain content to themselves learn better than those who do not

Creating content, as opposed to simply reading content, leads to improved recall

Learning and multiple choice questions?

- Self-explanation effect

In general, learners who explain content to themselves learn better than those who do not

- Generation effect

Creating content, as opposed to simply reading content, leads to improved recall

- Testing effect

Being tested on previously studied material improves recall

Learning and multiple choice questions?

- Self-explanation effect

In general, learners who explain content to themselves learn better than those who do not

- Generation effect

Creating content, as opposed to simply reading content, leads to improved recall

- Testing effect

Being tested on previously studied material improves recall

Bonus: you'll help future course participants learn.

Technicalities, you can use Markdown for questions, e.g.

````
Study the following `Dockerfile` configuration.

```
FROM denoland/deno:alpine-1.26.2
EXPOSE 7777
WORKDIR /app
COPY . .
RUN deno cache app.js
CMD [ "run", "--allow-n
```

Which of the following
functionality of the ab
````

Study the following `Dockerfile` configuration.

```
FROM denoland/deno:alpine-1.26.2
EXPOSE 7777
WORKDIR /app
COPY . .
RUN deno cache app.js
CMD [ "run", "--allow-net", "--watch", "app.js" ]
```

Which of the following options best describes the functionality of the above configuration?

Learning objectives?

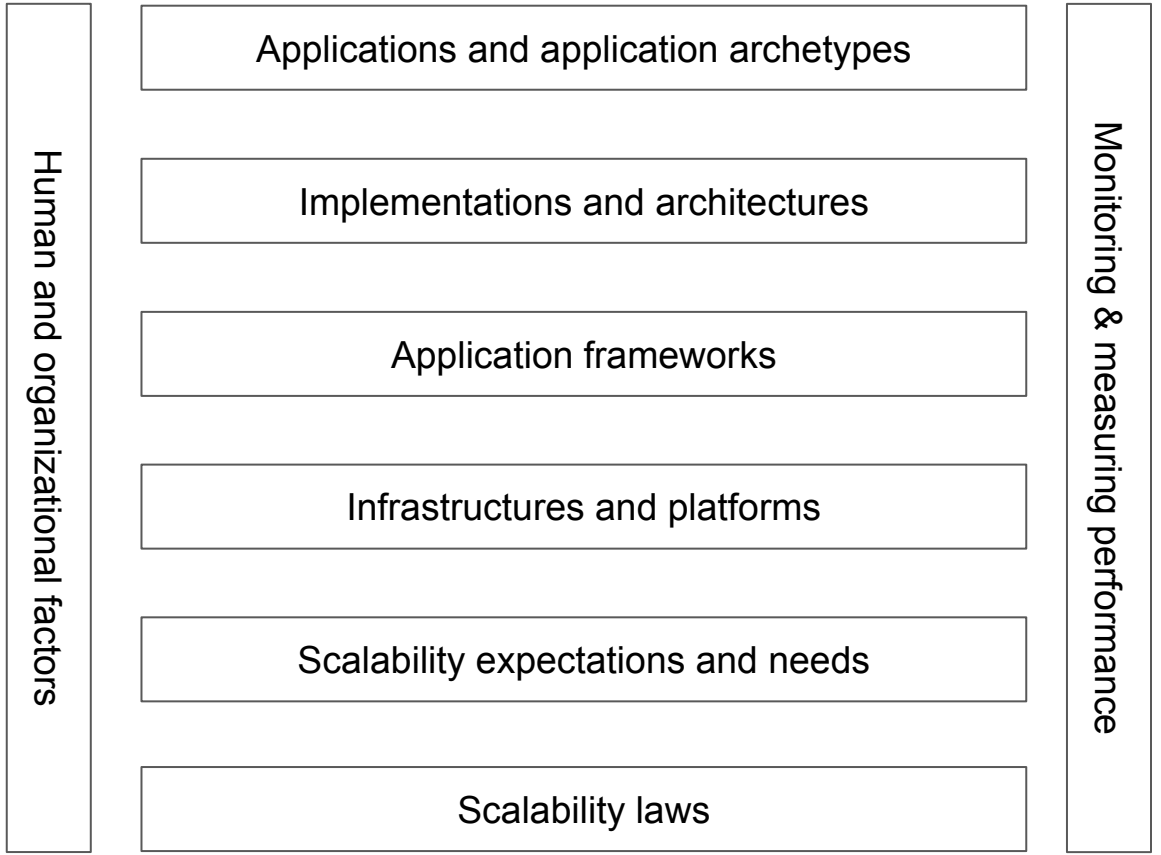
Learning objectives?

The course introduces learners to the principles of building scalable web applications, focusing on recent advances in both client- and server-side development as well as on platforms and hosting solutions. Architectural patterns and their fit and need for different types of web applications are also considered.

Learning objectives

- Understands the multiple dimensions of scalability and understands factors that contribute to the scalability of web applications.
- Knows and applies architectural patterns and techniques for designing and building scalable web applications.
- Understands the impact of the wide range of design decisions in building scalable web applications.
- Knows scalability laws and reflects on them in the context of architectural decisions.
- Can evaluate the scalability of web applications at multiple abstraction levels.
- Has practical experience in designing, building, and deploying web applications that scale.

The Big Picture



Sample readings (actual ones on platform)

Plenty of research on the course topic, see e.g.

- A scalable HTTP server: The NCSA prototype – https://doi.org/10.1016/0169-7552(94)90129-5
- Scalability issues for high performance digital libraries on the World Wide Web – https://doi.org/10.1109/ADL.1996.502524
- Enhancing the Web's infrastructure: From caching to replication – https://doi.org/10.1109/4236.601083
- Cluster-Based Scalable Network Services – https://dl.acm.org/doi/10.1145/268998.266662
- Globally Distributed Content Delivery – https://doi.org/10.1109/MIC.2002.1036038

Sample readings (actual ones on platform)

I intentionally only included articles that are over 20 years old into this list.

Plenty of research on the course topic, see e.g.

- A scalable HTTP server: The NCSA prototype – https://doi.org/10.1016/0169-7552(94)90129-5
- Scalability issues for high performance digital libraries on the World Wide Web – https://doi.org/10.1109/ADL.1996.502524
- Enhancing the Web's infrastructure: From caching to replication – https://doi.org/10.1109/4236.601083
- Cluster-Based Scalable Network Services – https://dl.acm.org/doi/10.1145/268998.266662
- Globally Distributed Content Delivery – https://doi.org/10.1109/MIC.2002.1036038

Sample readings (actual ones on platform)

I intentionally only included articles that are over 20 years old into this list.

Plenty of research on the course topic, see e.g.

- A scalable HTTP server: The NCSA prototype – https://doi.org/10.1016/0169-7552(94)90129-5
- Scalability issues for high performance digital libraries on the World Wide Web – https://doi.org/10.1109/ADL.1996.502524
- Enhancing the Web's infrastructure: From caching to replication – https://doi.org/10.1109/4236.601083
- Cluster-Based Scalable Network Services – https://dl.acm.org/doi/10.1145/268998.266662
- Globally Distributed Content Delivery – https://doi.org/10.1109/MIC.2002.1036038

But, there's also newer work!

Sample readings (actual ones on platform)

I intentionally only included articles that are over 20 years old into this list.

Plenty of research on the course topic, see e.g.

- A scalable HTTP server: The NCSA prototype – https://doi.org/10.1016/0169-7552(94)90129-5
- Scalability issues for high performance digital libraries on the World Wide Web – https://doi.org/10.1109/ADL.1996.502524
- Enhancing the Web's infrastructure: From caching to replication – https://doi.org/10.1109/4236.601083
- Cluster-Based Scalable Network Services – https://dl.acm.org/doi/10.1145/268998.266662
- Globally Distributed Content Delivery – https://doi.org/10.1109/MIC.2002.1036038
- The Google File System – https://dl.acm.org/doi/10.1145/945445.945450
- Dynamo: Amazon's highly available key-value store – https://doi.org/10.1109/MIC.2002.1036038

But, there's also newer work!

Sample readings (actual ones on platform)

I intentionally only included articles that are over 20 years old into this list.

Plenty of research on the course topic, see e.g.

- A scalable HTTP server: The NCSA prototype – https://doi.org/10.1016/0169-7552(94)90129-5
- Scalability issues for high performance digital libraries on the World Wide Web – https://doi.org/10.1109/ADL.1996.502524
- Enhancing the Web's infrastructure: From caching to replication – https://doi.org/10.1109/4236.601083
- Cluster-Based Scalable Network Services – https://dl.acm.org/doi/10.1145/268998.266662
- Globally Distributed Content Delivery – https://doi.org/10.1109/MIC.2002.1036038
- The Google File System – https://dl.acm.org/doi/10.1145/945445.945450
- Dynamo: Amazon's highly available key-value store – https://doi.org/10.1109/MIC.2002.1036038

But, there's also newer work!

Wait, these were also published over 15 years ago?

Actively studied area?

- New solutions are found for old problems as technologies evolve.
- New problems are identified as technologies evolve.
- Lots of open problems – a good and timely topic for BSc and MSc theses (also for PhD work! :))

Actively studied area?

- New solutions are found for old problems as technologies evolve.
- New problems are identified as technologies evolve.
- Lots of open problems – a good and timely topic for BSc and MSc theses (also for PhD work! :))

See e.g. “What serverless computing is and should become: the next phase of cloud computing” http://dx.doi.org/10.1145/3406011

Defining scalability

Defining scalability

(at least for now, other definitions will follow later on in the course)

Defining scalability

- “By *scalability* we mean that the proposed protocols for data delivery are cost-effective even when there are a very large number (100’s, 1000’s, even tens of thousands) of destinations that the data needs to be delivered to.” – Corona: A Communication Service for Scalable, Reliable Group Collaboration Systems (1996)

- “We call a system *scalable* if the system response time for individual requests is kept as small as theoretical possible when the number of simultaneous HTTP requests increases, while maintaining a low request drop rate and achieving a high peak request rate.” – SWEB: Towards a Scalable World Wide Web Server on MultiComputers (1996)

- “By *scalability*, we mean that when the load offered to the service increases, an incremental and linear increase in hardware can maintain the same per-user level of service” – Cluster-Based Scalable Network Services (1997)

Defining scalability

- “*Scalability* is a desirable attribute of a network, system, or process. The concept connotes the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement.” – Characteristics of Scalability and Their Impact on Performance (2000)

- “Systems are often said to be *scalable* if they present mechanisms for adding capacity as load increases.” – Characterizing the Scalability of a Large Web-Based Shopping System (2001)

- “*Scalability* means that Web service providers should be able to serve a fast-growing and unknown number of customers with minimal performance degradation.” – Capacity Planning: An Essential Tool for Managing Web Services (2002)

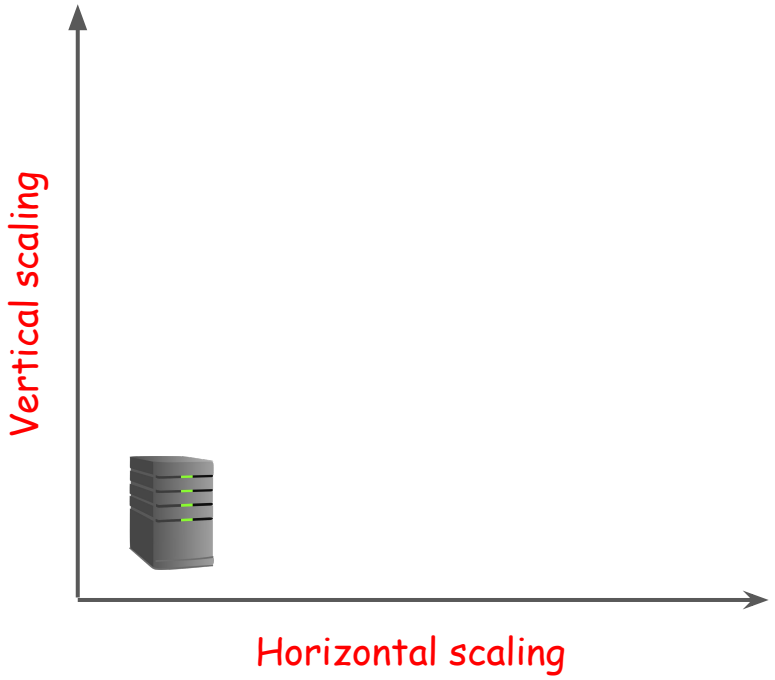
- “We consider a system to be *scalable* if there is a straightforward way to upgrade the system to handle an increase in traffic while maintaining adequate performance.” – Capacity Planning: An Essential Tool for Managing Web Services (2002)

Defining scalability

- Early on: concerns related to meeting increasing demands.

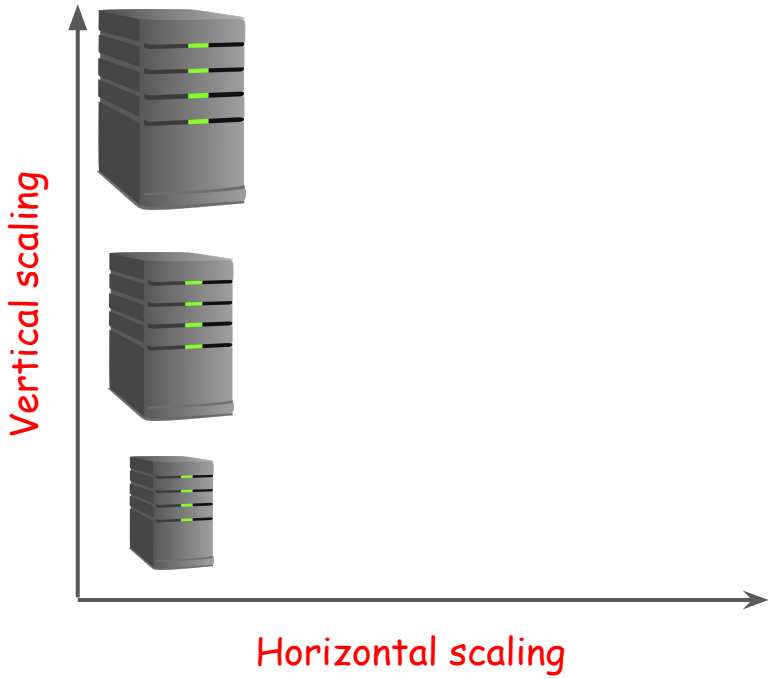
Defining scalability

- Vertical and horizontal scaling classically used as examples of *how* to scale



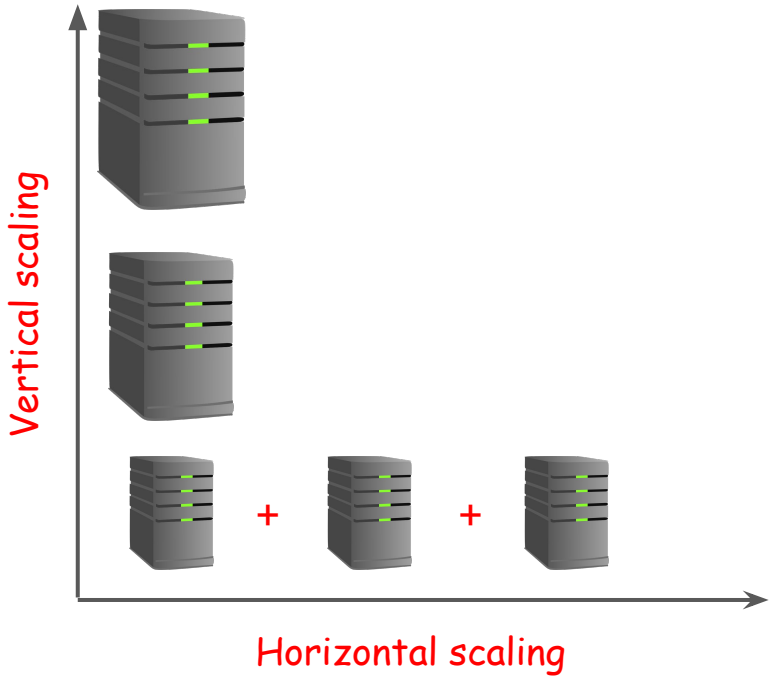
Defining scalability

- Vertical and horizontal scaling classically used as examples of *how* to scale
- Vertical scaling: scaling up – adding more resources



Defining scalability

- Vertical and horizontal scaling classically used as examples of *how* to scale
- Vertical scaling: scaling up – adding more resources
- Horizontal scaling: scaling out – adding more machines



Defining scalability

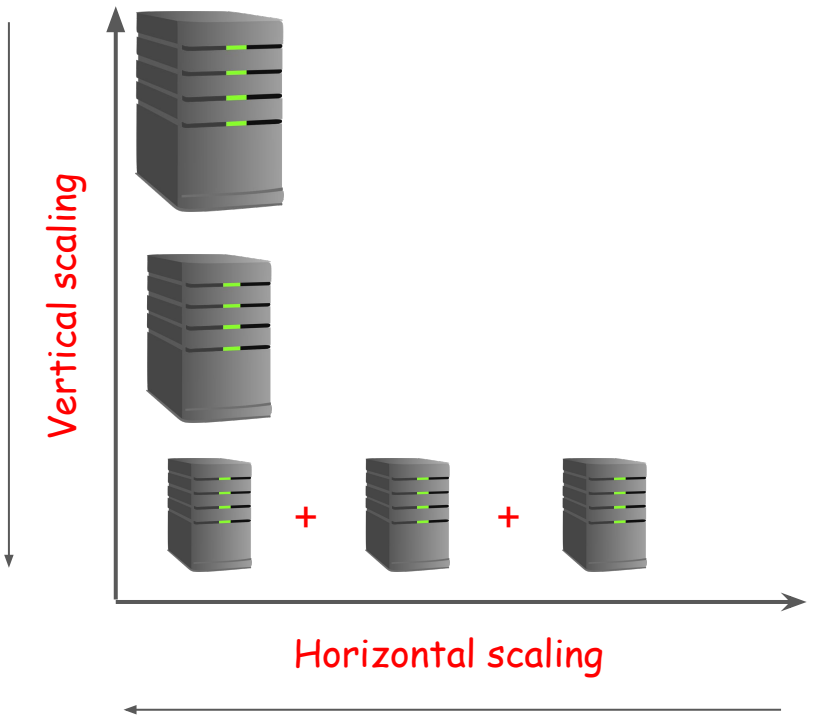
- Early on: concerns related to meeting increasing demands.

Defining scalability

- Early on: concerns related to meeting increasing demands.
- After a while: adding concerns about adjusting to fluctuating demands.

Defining scalability

- Vertical and horizontal scaling classically used as examples of *how* to scale
- Vertical scaling: scaling up – adding more resources
- Horizontal scaling: scaling out – adding more machines
- Also, scaling down and in!



CS-C3170 Web Software Development Recap

Materials at https://fitech101.aalto.fi/web-software-development/

Web Software Development

Client-server Model

Web Software Development

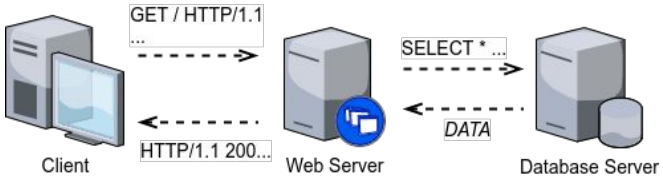
"Hello world!" application written for Deno

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts" ;

const handleRequest = (request) => {
  return new Response("Hello world!");
};

serve(handleRequest, { port: 7777 });
```

Web Software Development

Client-server Model

N-tier architecture: Sample with 3 tiers



Web Software Development

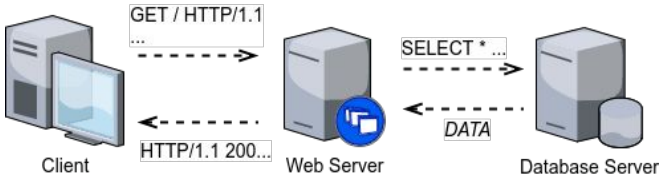
"Hello world!" application written for Deno

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts";
import { executeQuery } from "./database/database.js";

const handleRequest = async (request) => {
  const res = await executeQuery("SELECT COUNT(*) FROM table");
  return new Response(`Rows: ${res.rows[0].count}`);
};

serve(handleRequest, { port: 7777 });
```

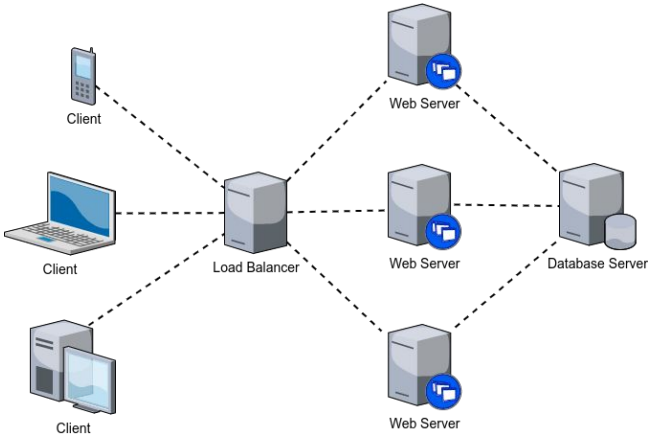
Web Software Development

Client-server Model

N-tier architecture: Sample with 3 tiers



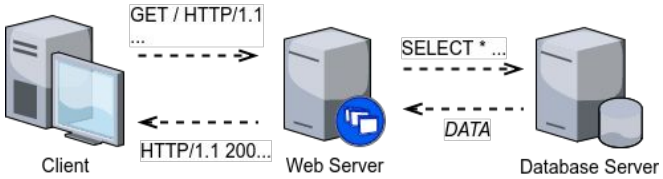
N-tier architecture: Sample with 4 tiers



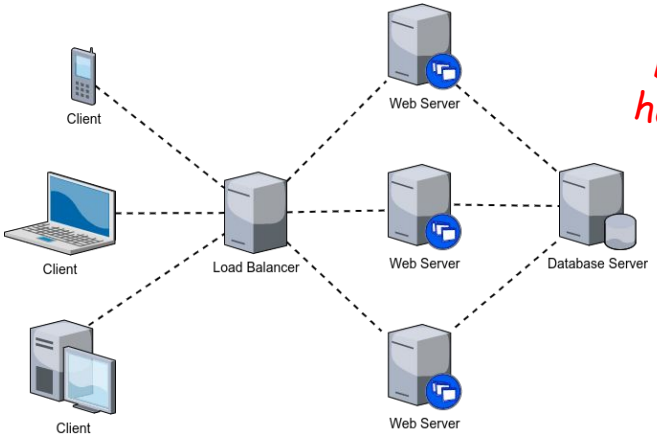
Web Software Development

Client-server Model

N-tier architecture: Sample with 3 tiers



N-tier architecture: Sample with 4 tiers



Did not really go here in CS-C3170

Measuring application performance

Measuring application performance

- Multiple performance metrics, including time to first byte, time to first paint, time to first contentful paint, time to interactive, etc.

Measuring application performance

- Multiple performance metrics, including time to first byte, time to first paint, time to first contentful paint, time to interactive, etc.

- For now, we're mostly interested in simpler HTTP performance statistics:
  - the number of HTTP requests that a server can handle per second
  - average response times (e.g. median, 95th percentile, 99th percentile, 99.9th percentile)
  - percentage of requests leading to errors

Measuring application performance

- Multiple performance metrics, including time to first byte, time to first paint, time to first contentful paint, time to interactive, etc.

- For now, we're mostly interested in simpler HTTP performance statistics:
  - the number of HTTP requests that a server can handle per second
  - average response times (e.g. median, 95th percentile, 99th percentile, 99.9th percentile)
  - percentage of requests leading to errors

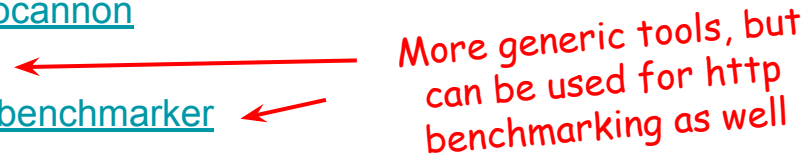
These, of course, under some stress :)

HTTP performance testing tools

- Good number of tools that can be used for benchmarking, including
  - Autocannon – https://github.com/mcollina/autocannon
  - Benny – https://github.com/caderek/benny
  - Deno bench – https://deno.land/manual/tools/benchmarker
  - Gatling – https://gatling.io/
  - JMeter – https://jmeter.apache.org/
  - k6 – https://k6.io/
  - wrk – https://github.com/wg/wrk and wrk2 – https://github.com/giltene/wrk2

HTTP performance testing tools

- Good number of tools that can be used for benchmarking, including
  - Autocannon – https://github.com/mcollina/autocannon
  - Benny – https://github.com/caderek/benny
  - Deno bench – https://deno.land/manual/tools/benchmarker
  - Gatling – https://gatling.io/
  - JMeter – https://jmeter.apache.org/
  - k6 – https://k6.io/
  - wrk – https://github.com/wg/wrk and wrk2 – https://github.com/giltene/wrk2



Example: k6

Example: k6

```
import { serve } from "https://deno.land/std@0.140.0/http/server.ts" ;

const handleRequest = (request) => {
  return new Response("Hello world!");
};

serve(handleRequest, { port: 7777 });
```

Testing a simple Hello world! application.

Example: k6

- Using k6 (https://k6.io/), we write a test script that is executed with k6

Example: k6

"do a GET request to this address"

- Using k6 (https://k6.io/), we write a test script that is executed with k6

```
import http from "k6/http";

export default function () {
  http.get("http://localhost:7777");
}
```

Example: k6

"do a GET request to this address"

- Using k6 (https://k6.io/), we write a test script that is executed with k6

```
k6 run script.js
```

```
import http from "k6/http";

export default function () {
  http.get("http://localhost:7777");
}
```

Example: k6

"do a GET request to this address"

- Using k6 (https://k6.io/), we write a test script that is executed with k6

```
k6 run script.js
```

```
import http from "k6/http";

export default function () {
  http.get("http://localhost:7777");
}
```

```
running (00m00.0s), 0/1 VUs, 1 complete and 0 interrupted iterations
default ✓ [======================================] 1 VUs  00m00.0s/10m0s  1/1 iters, 1 per VU

     data_received..................: 151 B 93 kB/s
     data_sent......................: 80 B  49 kB/s
     http_req_blocked...............: avg=191.44µs min=191.44µs med=191.44µs max=191.44µs p(90)=191.44µs p(95)=191.44µs
     http_req_connecting............: avg=93.11µs  min=93.11µs  med=93.11µs  max=93.11µs  p(90)=93.11µs  p(95)=93.11µs
     http_req_duration..............: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
       { expected_response:true }...: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
     http_req_failed................: 0.00% ✓ 0          ✗ 1
     http_req_receiving.............: avg=32.25µs  min=32.25µs  med=32.25µs  max=32.25µs  p(90)=32.25µs  p(95)=32.25µs
     http_req_sending...............: avg=39.61µs  min=39.61µs  med=39.61µs  max=39.61µs  p(90)=39.61µs  p(95)=39.61µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=184.37µs min=184.37µs med=184.37µs max=184.37µs p(90)=184.37µs p(95)=184.37µs
     http_reqs......................: 1     616.000237/s
     iteration_duration.............: avg=536.45µs min=536.45µs med=536.45µs max=536.45µs p(90)=536.45µs p(95)=536.45µs
     iterations.....................: 1     616.000237/s
```

Example: k6

- Using k6 (https://k6.io/), we write a test script that is executed with k6

```
k6 run script.js
```

```
import http from "k6/http";

export default function () {
  http.get("http://localhost:7777");
}
```

"do a GET request to this address"

```
running (00m00.0s), 0/1 VUs, 1 complete and 0 interrupted iterations
default ✓ [======================================] 1 VUs  00m00.0s/10m0s  1/1 iters, 1 per VU

     data_received..................: 151 B 93 kB/s
     data_sent......................: 80 B  49 kB/s
     http_req_blocked...............: avg=191.44µs min=191.44µs med=191.44µs max=191.44µs p(90)=191.44µs p(95)=191.44µs
     http_req_connecting............: avg=93.11µs  min=93.11µs  med=93.11µs  max=93.11µs  p(90)=93.11µs  p(95)=93.11µs
     http_req_duration..............: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
       { expected_response:true }...: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
     http_req_failed................: 0.00% ✓ 0          ✗ 1
     http_req_receiving.............: avg=32.25µs  min=32.25µs  med=32.25µs  max=32.25µs  p(90)=32.25µs  p(95)=32.25µs
     http_req_sending...............: avg=39.61µs  min=39.61µs  med=39.61µs  max=39.61µs  p(90)=39.61µs  p(95)=39.61µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=184.37µs min=184.37µs med=184.37µs max=184.37µs p(90)=184.37µs p(95)=184.37µs
     http_reqs......................: 1     616.000237/s
     iteration_duration.............: avg=536.45µs min=536.45µs med=536.45µs max=536.45µs p(90)=536.45µs p(95)=536.45µs
     iterations.....................: 1     616.000237/s
```

We made one request

Example: k6

- Using k6 (https://k6.io/), we write a test script that is executed with k6

```
k6 run script.js
```

```
import http from "k6/http";

export default function () {
  http.get("http://localhost:7777");
}
```

"do a GET request to this address"

```
running (00m00.0s), 0/1 VUs, 1 complete and 0 interrupted iterations
default ✓ [======================================] 1 VUs  00m00.0s/10m0s  1/1 iters, 1 per VU

     data_received..................: 151 B 93 kB/s
     data_sent......................: 80 B  49 kB/s
     http_req_blocked...............: avg=191.44µs min=191.44µs med=191.44µs max=191.44µs p(90)=191.44µs p(95)=191.44µs
     http_req_connecting............: avg=93.11µs  min=93.11µs  med=93.11µs  max=93.11µs  p(90)=93.11µs  p(95)=93.11µs
     http_req_duration..............: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
       { expected_response:true }...: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
     http_req_failed................: 0.00% ✓ 0          ✗ 1
     http_req_receiving.............: avg=32.25µs  min=32.25µs  med=32.25µs  max=32.25µs  p(90)=32.25µs  p(95)=32.25µs
     http_req_sending...............: avg=39.61µs  min=39.61µs  med=39.61µs  max=39.61µs  p(90)=39.61µs  p(95)=39.61µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=184.37µs min=184.37µs med=184.37µs max=184.37µs p(90)=184.37µs p(95)=184.37µs
     http_reqs......................: 1     616.000237/s
     iteration_duration.............: avg=536.45µs min=536.45µs med=536.45µs max=536.45µs p(90)=536.45µs p(95)=536.45µs
     iterations.....................: 1     616.000237/s
```

We made one request

Based on that, the server could handle 616 requests per second

Example: k6

"do a GET request to this address"

- Using k6 (https://k6.io/), we write a test script that is executed with k6

```
k6 run script.js
```

```
import http from "k6/http";

export default function () {
  http.get("http://localhost:7777");
}
```

```
running (00m00.0s), 0/1 VUs, 1 complete and 0 interrupted iterations
default ✓ [======================================] 1 VUs  00m00.0s/10m0s  1/1 iters, 1 per VU

     data_received..................: 151 B 93 kB/s
     data_sent......................: 80 B  49 kB/s
     http_req_blocked...............: avg=191.44µs min=191.44µs med=191.44µs max=191.44µs p(90)=191.44µs p(95)=191.44µs
     http_req_connecting............: avg=93.11µs  min=93.11µs  med=93.11µs  max=93.11µs  p(90)=93.11µs  p(95)=93.11µs
     http_req_duration..............: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
       { expected_response:true }...: avg=256.24µs min=256.24µs med=256.24µs max=256.24µs p(90)=256.24µs p(95)=256.24µs
     http_req_failed................: 0.00% ✓ 0          ✗ 1
     http_req_receiving.............: avg=32.25µs  min=32.25µs  med=32.25µs  max=32.25µs  p(90)=32.25µs  p(95)=32.25µs
     http_req_sending...............: avg=39.61µs  min=39.61µs  med=39.61µs  max=39.61µs  p(90)=39.61µs  p(95)=39.61µs
     http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=184.37µs min=184.37µs med=184.37µs max=184.37µs p(90)=184.37µs p(95)=184.37µs
     http_reqs......................: 1     616.000237/s
     iteration_duration.............: avg=536.45µs min=536.45µs med=536.45µs max=536.45µs p(90)=536.45µs p(95)=536.45µs
     iterations.....................: 1     616.000237/s
```

Example: k6

- Providing options to k6

Example: k6

- Providing options to k6

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

export default function () {
  http.get("http://localhost:7777");
}
```

Example: k6

- Providing options to k6

"Continue doing GET requests to this address for 5 seconds with 10 concurrent users"

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

export default function () {
  http.get("http://localhost:7777");
}
```

Example: k6

- Providing options to k6

```
k6 run script.js
```

"Continue doing GET requests to this address for 5 seconds with 10 concurrent users"

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

export default function () {
  http.get("http://localhost:7777");
}
```

Example: k6

- Providing options to k6

```
k6 run script.js
```

"Continue doing GET requests to this address for 5 seconds with 10 concurrent users"

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

ction () {
/localhost:7777" );
```

```
running (05.0s), 00/10 VUs, 114847 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  5s

     data_received..................: 17 MB  3.5 MB/s
     data_sent......................: 9.2 MB 1.8 MB/s
     http_req_blocked...............: avg=2.16µs   min=644ns   med=1.89µs   max=7.29ms   p(90)=2.82µs   p(95)=3.05µs
     http_req_connecting............: avg=5ns      min=0s      med=0s       max=144.76µs p(90)=0s       p(95)=0s
     http_req_duration..............: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
       { expected_response:true }...: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
     http_req_failed................: 0.00%  ✓ 0           ✗ 114847
     http_req_receiving.............: avg=22.88µs  min=5.18µs  med=21.6µs   max=7.87ms   p(90)=29.53µs  p(95)=33.41µs
     http_req_sending...............: avg=9.36µs   min=2.91µs  med=8.42µs   max=3.12ms   p(90)=13.99µs  p(95)=16.1µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=352.83µs min=53.13µs med=370.38µs max=5.67ms   p(90)=441.84µs p(95)=461.92µs
     http_reqs......................: 114847 22963.165726/s
     iteration_duration.............: avg=427.72µs min=93.28µs med=449.25µs max=8.75ms   p(90)=521.43µs p(95)=544.71µs
     iterations.....................: 114847 22963.165726/s
     vus............................: 10     min=10        max=10
     vus_max........................: 10     min=10        max=10
```

Example: k6

- Providing options to k6

```
k6 run script.js
```

We made 114847 requests

```
running (05.0s), 00/10 VUs, 114847 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  5s

     data_received..................: 17 MB  3.5 MB/s
     data_sent......................: 9.2 MB 1.8 MB/s
     http_req_blocked...............: avg=2.16µs   min=644ns   med=1.89µs   max=7.29ms   p(90)=2.82µs   p(95)=3.05µs
     http_req_connecting............: avg=5ns      min=0s      med=0s       max=144.76µs p(90)=0s       p(95)=0s
     http_req_duration..............: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
       { expected_response:true }...: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
     http_req_failed................: 0.00%  ✓ 0           ✗ 114847
     http_req_receiving.............: avg=22.88µs  min=5.18µs  med=21.6µs   max=7.87ms   p(90)=29.53µs  p(95)=33.41µs
     http_req_sending...............: avg=9.36µs   min=2.91µs  med=8.42µs   max=3.12ms   p(90)=13.99µs  p(95)=16.1µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=352.83µs min=53.13µs med=370.38µs max=5.67ms   p(90)=441.84µs p(95)=461.92µs
     http_reqs......................: 114847 22963.165726/s
     iteration_duration.............: avg=427.72µs min=93.28µs med=449.25µs max=8.75ms   p(90)=521.43µs p(95)=544.71µs
     iterations.....................: 114847 22963.165726/s
     vus............................: 10     min=10        max=10
     vus_max........................: 10     min=10        max=10
```

"Continue doing GET requests to this address for 5 seconds with 10 concurrent users"

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

ction () {
/localhost:7777");
```

Example: k6

- Providing options to k6

```
k6 run script.js
```

"Continue doing GET requests to this address for 5 seconds with 10 concurrent users"

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

ction () {
/localhost:7777");
```

We made 114847 requests

Based on that, the server could handle 22963 requests per second

```
running (05.0s), 00/10 VUs, 114847 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  5s

     data_received..................: 17 MB  3.5 MB/s
     data_sent......................: 9.2 MB 1.8 MB/s
     http_req_blocked...............: avg=2.16µs   min=644ns   med=1.89µs   max=7.29ms   p(90)=2.82µs   p(95)=3.05µs
     http_req_connecting............: avg=5ns      min=0s      med=0s       max=144.76µs p(90)=0s       p(95)=0s
     http_req_duration..............: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
       { expected_response:true }...: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
     http_req_failed................: 0.00%  ✓ 0           ✗ 114847
     http_req_receiving.............: avg=22.88µs  min=5.18µs  med=21.6µs   max=7.87ms   p(90)=29.53µs  p(95)=33.41µs
     http_req_sending...............: avg=9.36µs   min=2.91µs  med=8.42µs   max=3.12ms   p(90)=13.99µs  p(95)=16.1µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=352.83µs min=53.13µs med=370.38µs max=5.67ms   p(90)=441.84µs p(95)=461.92µs
     http_reqs......................: 114847 22963.165726/s
     iteration_duration.............: avg=427.72µs min=93.28µs med=449.25µs max=8.75ms   p(90)=521.43µs p(95)=544.71µs
     iterations.....................: 114847 22963.165726/s
     vus............................: 10     min=10         max=10
     vus_max........................: 10     min=10         max=10
```

Example: k6

- Providing options to k6

```
k6 run script.js
```

"Continue doing GET requests to this address for 5 seconds with 10 concurrent users"

```
import http from "k6/http";

export const options = {
  duration: "5s",
  vus: 10,
};

ction () {
/localhost:7777");
```

We made 114847 requests

Based on that, the server could handle 22963 requests per second

```
running (05.0s), 00/10 VUs, 114847 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  5s

     data_received..................: 17 MB  3.5 MB/s
     data_sent......................: 9.2 MB 1.8 MB/s
     http_req_blocked...............: avg=2.16µs   min=644ns   med=1.89µs   max=7.29ms   p(90)=2.82µs   p(95)=3.05µs
     http_req_connecting............: avg=5ns      min=0s      med=0s       max=144.76µs p(90)=0s       p(95)=0s
     http_req_duration..............: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
       { expected_response:true }...: avg=385.07µs min=70.32µs med=404.1µs  max=8.32ms   p(90)=475.51µs p(95)=496.77µs
     http_req_failed................: 0.00%  ✓ 0           ✗ 114847
     http_req_receiving.............: avg=22.88µs  min=5.18µs  med=21.6µs   max=7.87ms   p(90)=29.53µs  p(95)=33.41µs
     http_req_sending...............: avg=9.36µs   min=2.91µs  med=8.42µs   max=3.12ms   p(90)=13.99µs  p(95)=16.1µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=352.83µs min=53.13µs med=370.38µs max=5.67ms   p(90)=441.84µs p(95)=461.92µs
     http_reqs......................: 114847 22963.165726/s
     iteration_duration.............: avg=427.72µs min=93.28µs med=449.25µs max=8.75ms   p(90)=521.43µs p(95)=544.71µs
     iterations.....................: 114847 22963.165726/s
     vus............................: 10     min=10         max=10
     vus_max........................: 10     min=10         max=10
```

Results can vary between runs - do not pick the best run :)..

Versions, technologies, etc matter a bit

Versions, technologies, etc matter a bit

The previous results with Deno 1.21.0 and library version @1.40.0

Versions, technologies, etc matter a bit

The previous results with Deno 1.21.0 and library version @1.40.0

Let’s try the same with Deno 1.26.2 and Deno’s flash Server (introduced in 1.25). Code:

```
Deno.serve((_) => new Response("Hello world!"), { port: 7777 });
```

Versions, technologies, etc matter a bit

The previous results with Deno 1.21.0 and library version @1.40.0

Let's try the same with Deno 1.26.2 and Deno's flash Server (introduced in 1.25). Code:

```
Deno.serve((_) => new Response("Hello world!"), { port: 7777 });
```

Run with: `deno run --allow-net --unstable app.js`

Same test: `k6 run script.js`

Versions, technologies, etc matter a bit

The previous results with Deno 1.21.0 and library version @1.40.0

Let's try the same with Deno 1.26.2 and Deno's flash Server (introduced in 1.25). Code:

```
Deno.serve((_) => new Response("Hello world!"), { port: 7777 });
```

Run with: `deno run --allow-net --unstable app.js`

Same test: `k6 run script.js`

```
running (05.0s), 00/10 VUs, 213893 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  5s

     data_received..................: 27 MB  5.5 MB/s
     data_sent......................: 17 MB  3.4 MB/s
     http_req_blocked...............: avg=2.25µs   min=658ns   med=2.2µs    max=7.41ms   p(90)=2.87µs   p(95)=3.19µs
     http_req_connecting............: avg=3ns      min=0s      med=0s       max=132.98µs p(90)=0s       p(95)=0s
     http_req_duration..............: avg=180.28µs min=32.54µs med=144.93µs max=12.43ms  p(90)=280.71µs p(95)=319.76µs
       { expected_response:true }...: avg=180.28µs min=32.54µs med=144.93µs max=12.43ms  p(90)=280.71µs p(95)=319.76µs
     http_req_failed................: 0.00%  ✓ 0           ✗ 213893
     http_req_receiving.............: avg=22.4µs   min=4.93µs  med=22.36µs  max=11.85ms  p(90)=28.07µs  p(95)=32.71µs
     http_req_sending...............: avg=9.69µs   min=2.64µs  med=9.67µs   max=8.4ms    p(90)=11.97µs  p(95)=13.36µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=148.18µs min=20.29µs med=113.11µs max=8.31ms   p(90)=241.61µs p(95)=277.55µs
     http_reqs......................: 213893 42768.681583/s
     iteration_duration.............: avg=225.56µs min=48.15µs med=193.73µs max=13.03ms  p(90)=335.54µs p(95)=380.9µs
     iterations.....................: 213893 42768.681583/s
     vus............................: 10     min=10        max=10
     vus_max........................: 10     min=10        max=10
```

Now, 42679 requests per second

Versions, technologies, etc matter a bit

The previous results with Deno 1.21.0 and library version @1.40.0

Let's try the same with Deno 1.26.2 and Deno's flash Server (introduced in 1.25). Code:

```
Deno.serve((_) => new Response("Hello world!"), { port: 7777 });
```

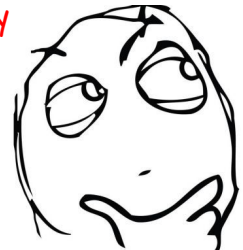
Run with: `deno run --allow-net --unstable app.js`

Same test: `k6 run script.js`

```
running (05.0s), 00/10 VUs, 213893 complete and 0 interrupted iterations
default ✓ [======================================] 10 VUs  5s

     data_received..................: 27 MB  5.5 MB/s
     data_sent......................: 17 MB  3.4 MB/s
     http_req_blocked...............: avg=2.25µs   min=658ns   med=2.2µs    max=7.41ms   p(90)=2.87µs   p(95)=3.19µs
     http_req_connecting............: avg=3ns      min=0s      med=0s       max=132.98µs p(90)=0s       p(95)=0s
     http_req_duration..............: avg=180.28µs min=32.54µs med=144.93µs max=12.43ms  p(90)=280.71µs p(95)=319.76µs
       { expected_response:true }...: avg=180.28µs min=32.54µs med=144.93µs max=12.43ms  p(90)=280.71µs p(95)=319.76µs
     http_req_failed................: 0.00%  ✓ 0           ✗ 213893
     http_req_receiving.............: avg=22.4µs   min=4.93µs  med=22.36µs  max=11.85ms  p(90)=28.07µs  p(95)=32.71µs
     http_req_sending...............: avg=9.69µs   min=2.64µs  med=9.67µs   max=8.4ms    p(90)=11.97µs  p(95)=13.36µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=148.18µs min=20.29µs med=113.11µs max=8.31ms   p(90)=241.61µs p(95)=277.55µs
     http_reqs......................: 213893 42768.681583/s
     iteration_duration.............: avg=225.56µs min=48.15µs med=193.73µs max=13.03ms  p(90)=335.54µs p(95)=380.9µs
     iterations.....................: 213893 42768.681583/s
     vus............................: 10     min=10        max=10
     vus_max........................: 10     min=10        max=10
```

Now, 42679 requests per second

Does "Hello world!" performance really matter?



First week readings

First week readings

First week readings contain a rehearsal of the Web Software Development course (a prerequisite for this course).

You'll get to rehearse the materials and to come up with questions on web development.

First course project

First course project: comparing implementations

- In the first course project, your task is to create **bit.ly**-like implementations, compare their performance, and write a report of the results.

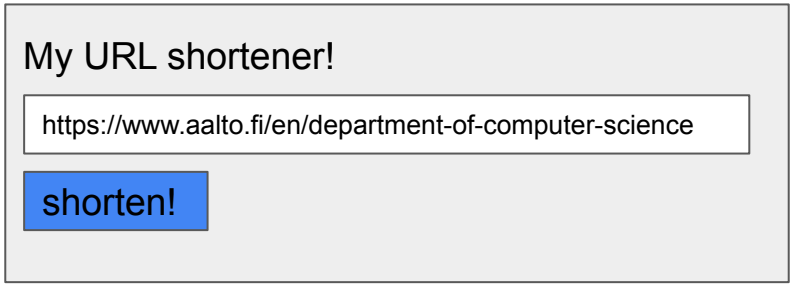
First course project: comparing implementations

- In the first course project, your task is to create **bit.ly**-like implementations, compare their performance, and write a report of the results.
- Each implementation should feature:
  - A main page that has a form into which users can write URLs that need to be shortened.
  - A database that stores the URLs and their shortened versions – entering an URL into the form and submitting it through the form stores the URL into the database and returns a page that contains a shortened version of the URL.
  - When a user accesses a shortened version of the URL, the server returns a response that redirects the user to the new URL.

First course project

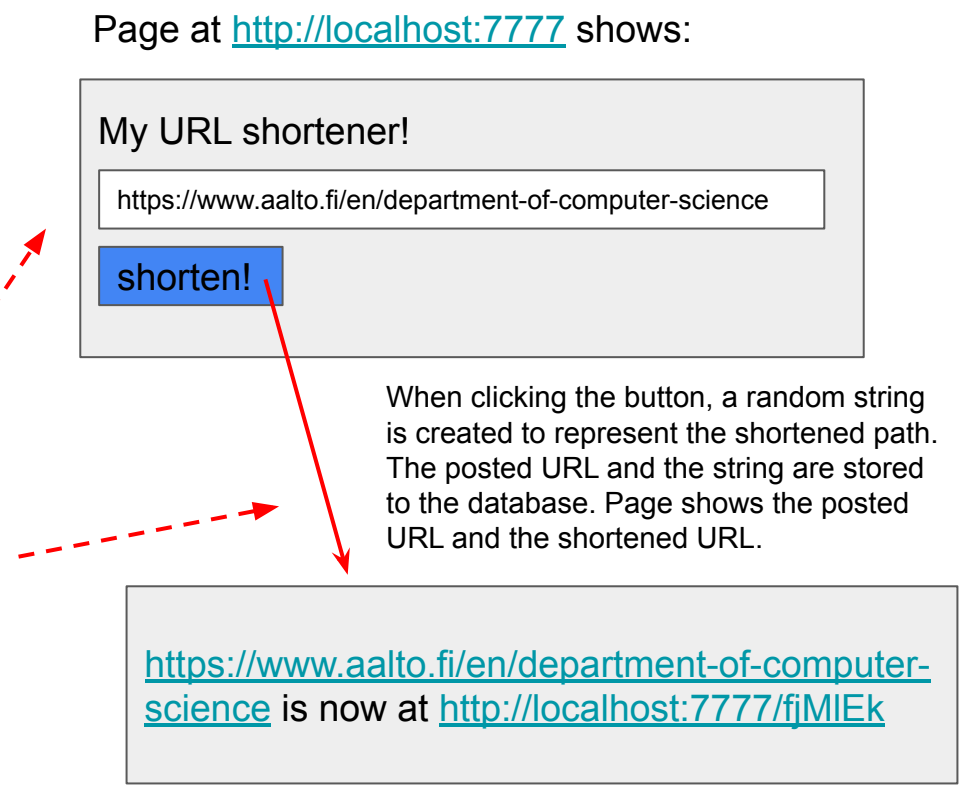
- In the first course project, your task is to create **bit.ly**-like implementations, compare their performance, and write a report of the results.
- Each implementation should feature:
  - A main page that has a form into which users can write URLs that need to be shortened.
  - A database that stores the URLs and their shortened versions – entering an URL into the form and submitting it through the form stores the URL into the database and returns a page that contains a shortened version of the URL.
  - When a user accesses a shortened version of the URL, the server returns a response that redirects the user to the new URL.

Page at http://localhost:7777 shows:



First course project

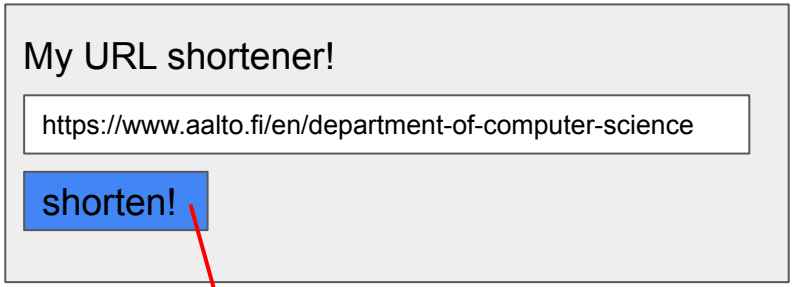
- In the first course project, your task is to create **bit.ly**-like implementations, compare their performance, and write a report of the results.
- Each implementation should feature:
  - A main page that has a form into which users can write URLs that need to be shortened.
  - A database that stores the URLs and their shortened versions – entering an URL into the form and submitting it through the form stores the URL into the database and returns a page that contains a shortened version of the URL.
  - When a user accesses a shortened version of the URL, the server returns a response that redirects the user to the new URL.



First course project

- In the first course project, your task is to create **bit.ly**-like implementations, compare their performance, and write a report of the results.
- Each implementation should feature:
  - A main page that has a form into which users can write URLs that need to be shortened.
  - A database that stores the URLs and their shortened versions – entering an URL into the form and submitting it through the form stores the URL into the database and returns a page that contains a shortened version of the URL.
  - When a user accesses a shortened version of the URL, the server returns a response that redirects the user to the new URL.

Page at http://localhost:7777 shows:



When clicking the button, a random string is created to represent the shortened path. The posted URL and the string are stored to the database. Page shows the posted URL and the shortened URL.

https://www.aalto.fi/en/department-of-computer-science is now at http://localhost:7777/fjMlEk

Now, accessing http://localhost:7777/fjMlEk redirects the user to https://www.aalto.fi/en/department-of-computer-science

First course project - passing requirements

- Two implementations done using the same programming language but a different framework (e.g. vanilla Deno vs. Oak, vanilla NodeJS vs Express, FastAPI vs Flask).
- Using a relational database (e.g. PostgreSQL).
- Performance tests for (1) the main page, (2) submitting the form to the database, and (3) asking for redirection. In the tests, record the average requests per second and the median, 95th percentile, and 99th percentile HTTP request duration. Run the tests with a sensible number of concurrent users for at least 10 seconds.
- All implementations and performance test scripts returned in a format that allows running them easily locally on Windows, Linux and Mac (i.e. a docker-compose configuration or similar for running the applications; performance test scripts for performance tests).
- Summary report with comparison results.

First course project - passing requirements

- Two implementations done using the same programming language but a different framework (e.g. vanilla Deno vs. Oak, vanilla NodeJS vs Express, FastAPI vs Flask).
- Using a relational database (e.g. PostgreSQL).
- Performance tests for (1) the main page, (2) submitting the form to the database, and (3) asking for redirection. In the tests, record the average requests per second and the median, 95th percentile, and 99th percentile HTTP request duration. Run the tests with a sensible number of concurrent users for at least 10 seconds.
- All implementations and performance test scripts returned in a format that allows running them easily locally on Windows, Linux and Mac (i.e. a docker-compose configuration or similar for running the applications; performance test scripts for performance tests).
- Summary report with comparison results.

Similar to the Web Software Development course, there is a project starter (Walking Skeleton) template that can be used to start the project with.

First course project - passing requirements / report

- A markdown-formatted document (no binary content) with:
  - Brief guidelines for running the applications and the performance tests.
  - Results of 6 performance tests (2 implementations times 3 performance tests). In the results, include the average requests per second and the median, 95th percentile, and 99th percentile HTTP request duration.
  - A brief reflection (5-10 sentences) on the reasons for possible performance differences between the pages and between the implementations.

First course project - passing with merits

- In addition to fulfilling the passing requirements:
  - A third implementation written in a different (non superset / subset) programming language (e.g. typescript and javascript do not count as different languages, while Python and javascript do).
  - Additional functionality: the user can ask to be redirected to a random location (out of the possibilities already in the database). This behavior is at the path `/random` of the application. That is, accessing the path http://localhost:7777/random redirects the user to a randomly picked address.
  - Performance tests for the additional functionality.
  - The report now with results of 12 performance tests (3 implementations times 4 performance tests). In the results, include the average requests per second and the median, 95th percentile, and 99th percentile HTTP request duration.
  - Brief suggestions for improving the performance of the applications (5-10 sentences).