

L8: Backend Abstractions

Web Engineering

188.951 2VU SS20

Jürgen Cito

L8: Backend Abstractions

- Overview of abstractions that enable building backends of web services
- Case study demonstrating these abstractions in Node.js/Express framework
- Brief overview of general programming abstractions commonly used when building web service backends

Learning Goals

- Get an overview of important concepts in backend web development
- Understand the difference between abstractions and concepts vs. concrete executions and implementations
- Ability to map these concepts and abstractions to their concrete counterparts in Node.js and the Express framework

Recap: Webserver

Web Server: Program running on a computer/server that accepts HTTP requests over a specific port and answers with HTTP responses

Basic web server in Node.js

```
const http = require('http');

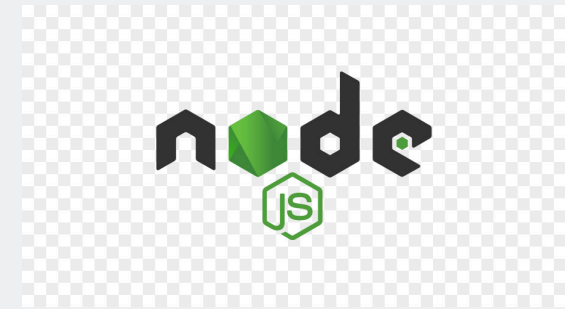
const requestListener = function (req, res) {
  res.writeHead(200);
  res.end('Hello, World!');
}

const server = http.createServer(requestListener);
server.listen(8080);
```

What are the essential **building blocks** to enable us to **efficiently** build web service **backends**?

Backend Abstractions: Case Study in Node/Express

Node.js is a JavaScript runtime environment that runs Chrome's V8 engine outside of the browser. It is event-driven (listening for requests) and provides facilities for synchronous and asynchronous computation



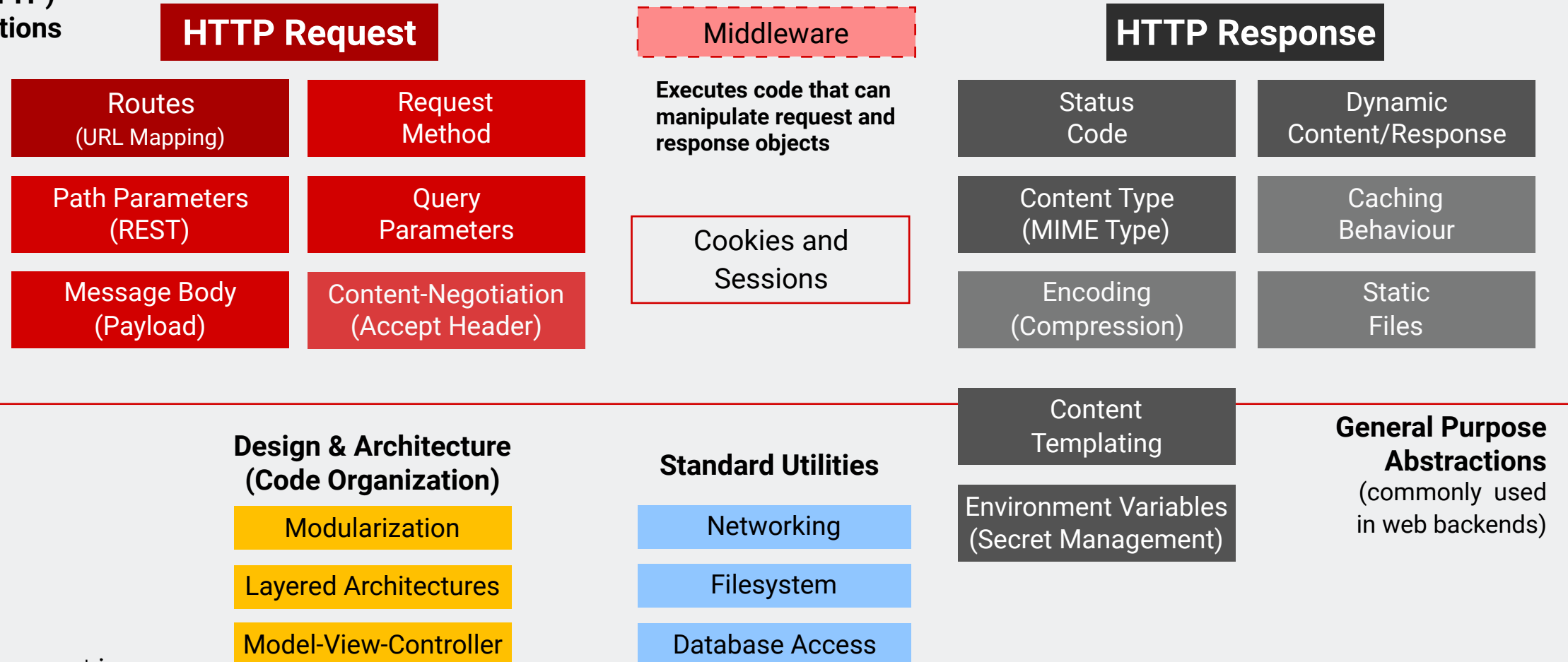
NPM (Node Package Manager) manages dependencies of external JavaScript packages, hosted in a package repository called npm registry



Express.js is a web framework for Node.js that provides backend abstractions

Concepts and Abstractions for Web Service Backends

Web (HTTP) Abstractions



Routes (URL Mapping)

Backend Abstractions

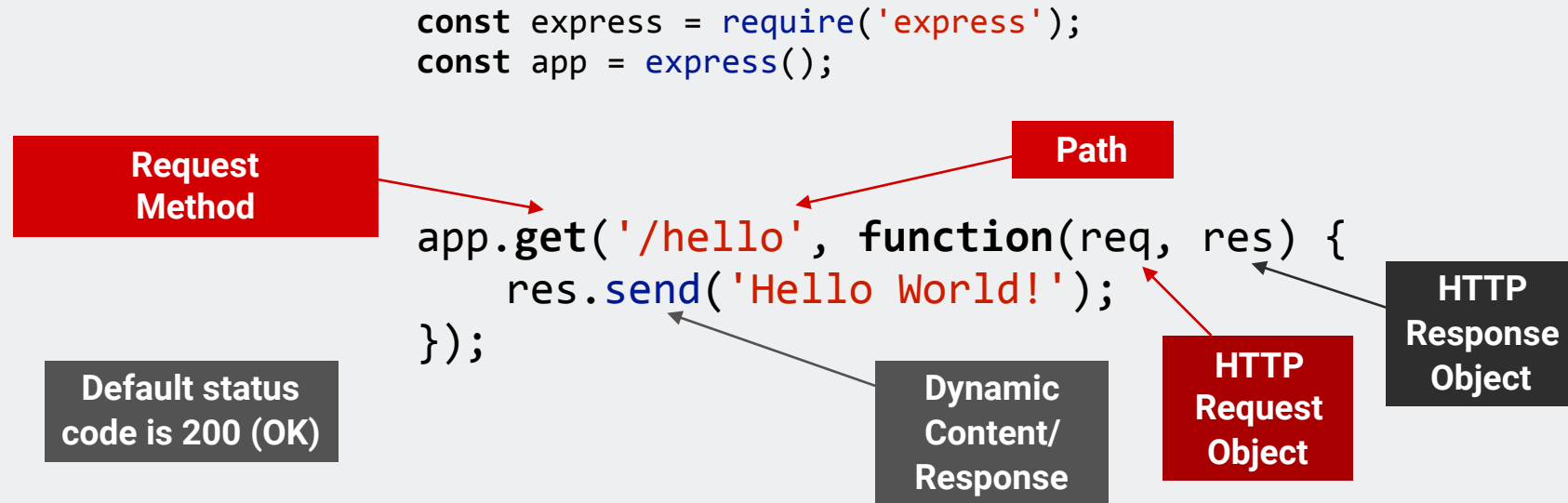
```
const express = require('express');  
const app = express();
```

```
app.get('/hello', function(req, res) {  
    res.send('Hello World!');  
});
```

```
const port = 3000;  
app.listen(port, function() {  
    console.log(`Waiting for requests on Port ${port}!`);  
});
```

Routes (URL Mapping)

Backend Abstractions



```
const port = 3000;
app.listen(port, function() {
  console.log(`Waiting for requests on Port ${port}!`);
});
```


Request and Response Objects

Backend Abstractions

```
app.put('/recipes/:id', (req, res) => {  
  const recipeId = req.params.id;  
  const hasImage = req.query.hasImage == 'true';  
  
  const recipe = Recipes.find(recipeId, hasImage);  
  if(!recipe) {  
    return res.sendStatus(404);  
  }  
  
  const payload = req.body;  
  recipe.update(payload);  
  res.send({updateSuccess : recipeId});  
});
```

Request and Response Objects

Backend Abstractions

```
app.put('/recipes/:id', (req, res) => {  
  const recipeId = req.params.id;  
  const hasImage = req.query.hasImage == 'true';  
  const recipe = Recipes.find(recipeId, hasImage);  
  if(!recipe) {  
    return res.sendStatus(404);  
  }  
  const payload = req.body;  
  recipe.update(payload);  
  res.send({updateSuccess : recipeId});  
});
```

Path Parameters

Query Parameters

Status Code set in response header and message body (for response header only see `res.status`)

Message Body as structured object (key-value pairs)

JavaScript objects are automatically serialized as JSON when sending the response. Could also use `res.json(obj)`

Content type inferred - but could also be set with `res.type('application/json')`

Middleware

Backend Abstractions

Middleware functions can manipulate request and response objects for every request-response cycle. They are also provided a **next()** function that invokes the next middleware function in the chain (order matters)

```
// for parsing application/json
app.use(express.json());
// for parsing HTML form data
// application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));
...
const payload = req.body;
...
```

**Message Body
as structured object
(key-value pairs)**

Message body only possible because middleware intercepted the request, classified and parsed the message, and then set `req.body`

Cookies and Sessions

Backend Abstractions

Cookies are the consequence of the stateless nature of the HTTP protocol paired with the desire of still establishing some notion of association between client and server.

Cookies can be set by both client and server as part of HTTP headers and are transmitted with every request/response cycle

Sessions use cookies to store a unique identifier. The associated session data is stored on the server (either in-memory or in persistent storage)

```
routes.get('/', async (req, res) => {  
  let sessionId = req.cookies.sessionId;  
  ...  
  if(!sessionId) {  
    ...  
    res.cookie('sessionId', sessionId);  
  }  
  ...  
}
```

Cookies parsed through middleware from request header

Cookie written into response header

Environment Variables & Secret Management

Backend Abstractions

Environment variables provide a standard way for configurability and provide a strict way of separating configuration from code. There are also several other ways to pass configuration to the program (pass parameters, read from configuration file, etc.)

```
const port = process.argv.length >= 3 ? +process.argv[2] : 3000;
```

Pass as parameter
to process

```
const db = require('db')
db.connect({
  host: process.env.DB_HOST,
  username: process.env.DB_USER,
  password: process.env.DB_PASS
})
```

Parameters come
from environment
provided by the
operating system

Templating

Backend Abstractions

Templates (sometimes also called views) provide separation between program logic and output.

Template engines replace variables in static template files and control structures (conditionals and loops) with values passed from the program.

```
app.set('view engine', 'pug')
...
routes.get('/', async (req, res) => {
  res.render('users', { title: 'Users',
    heading: 'List of users', users: getUsers() });
})
```

Enabled by
middleware concept

PUG Template - users.pug

```
html
  head
    title= title
  body
    h1= heading
    div#container
      - for user in users
        div.user= user.email
```

Output for rendered response

```
<html>
  <head>
    <title>Users</title>
  </head>
  <h1>List of users</h1>
  <div id="container">
    <div class="user">
      jane.doe@tuwien.ac.at
    </div>
    <div class="user">
      jack.bauer@tuwien.ac.at
    </div>
  </div>
</html>
```

Networking (HTTP)

Standard Utilities

Almost every programming language has multiple libraries of dealing with network and HTTP requests. Node also has `node-fetch`, that has the same functionality and familiar contract as the one in the browser API.

```
const fetch = require('node-fetch');
...
const response = await fetch(objectRequestUrl(objectID));
if(response.status !== 200) {
    console.log('Could not find object with id' + objectID);
    return false;
}
const object = await response.json();
```

Persistent Storage (Files)

Standard Utilities

```
const fs = require('fs');  
const path = require('path');
```

Filesystem utilities have
synchronous and
asynchronous API in Node

```
const destinations = JSON.parse(fs.readFileSync(path.join(__dirname, '../res/data.json')));
```

Beware when deploying to the cloud: Writing to the local filesystem on a server can lead to data loss if the server is ephemeral (as many platform-as-a-service (PaaS) cloud offerings are). The same goes for “local” databases.

Use so-called backing services for attached resources you can access from an API for persistent storage.

<https://12factor.net/backing-services>

Modules

Code Organization

Modules in Node.js are **not the same** as ES6 Modules we have seen for JavaScript in the browser. But in similar ways, it enables code organization through file-based separation and encapsulation

routes/artworks.js

Relative Path

```
const met = require('../utils/met.js');  
const artworks = met.search('van gogh');
```

Everything in module files not
in `module.exports` is
private/implementation detail

util/met.js

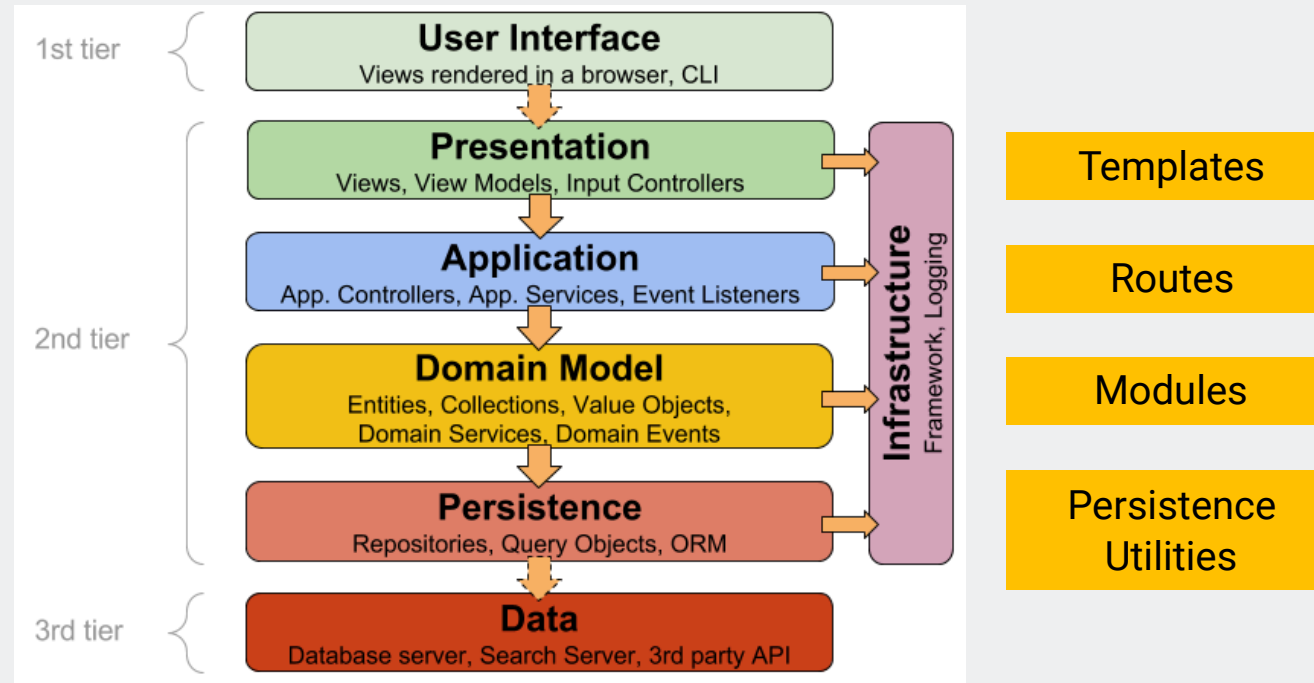
```
const search = async (term, max=100) => { ... }  
...  
module.exports.search = search;
```

Elements of `module.exports`
become part of `met` object

Layered Architectures

Common Web Architectures

Layering in web service backends can be facilitated through existing abstractions



<https://herbertograca.com/2017/08/03/layered-architecture/>

Discussion: Limitations of backend abstraction view

- Do libraries and APIs provided in other languages/web frameworks adhere to this view? What are the differences?
- Is this view future-proof? Why or why not?

