SEBA Master: Web Application Engineering

6. Building a REST-enabled Backend Service

Prof. Dr. Florian Matthes, SS21

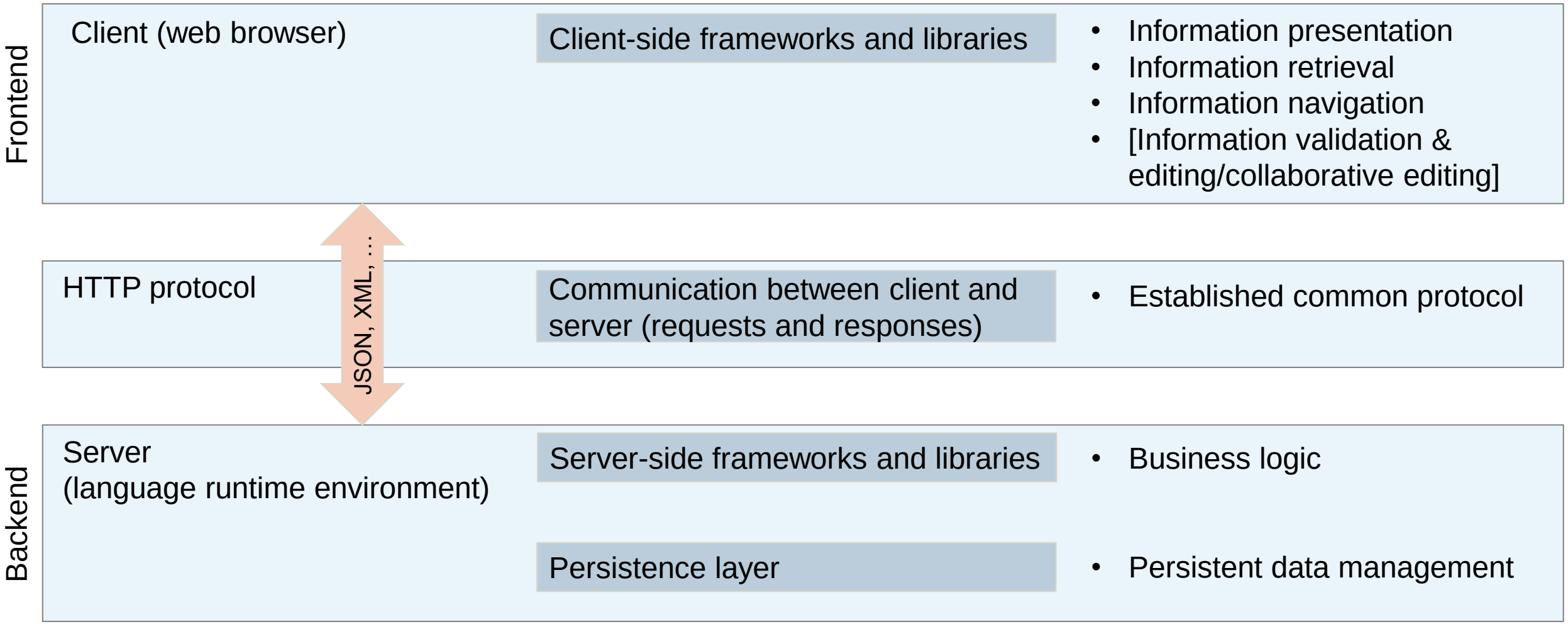
Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

Outline of the Lecture

6. Building a REST-enabled Backend Service

- Target Architecture and Development Environment
- Event-driven Architecture and Asynchronous I/O Operations with NodeJS
- Creating REST Interfaces Using Express
- Using Document-oriented Database Storage: MongoDB Example
- Enabling User Authentication on the Web Service Using JSON Web Tokens (JWT)

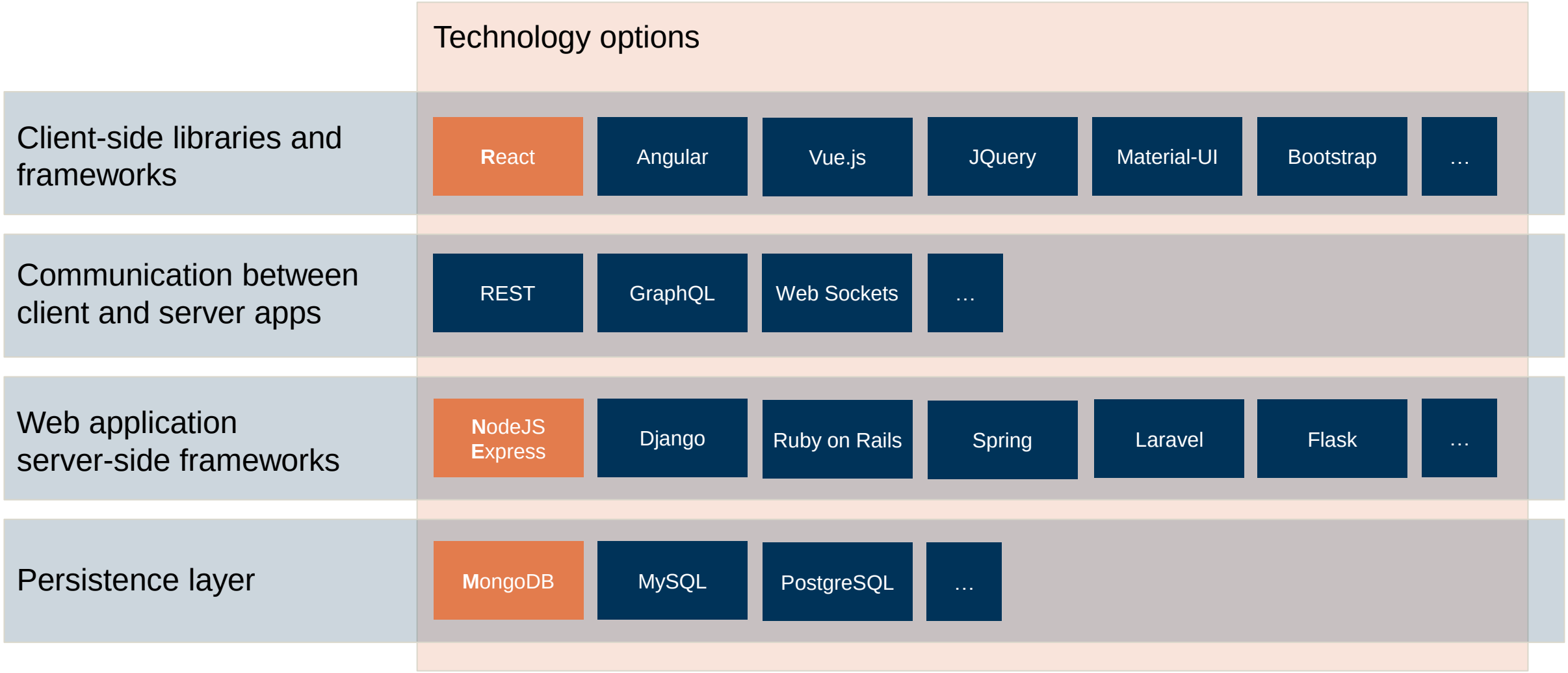
Architecture of Web Applications



Continuously Evolving Technology Options for Web Applications

Example of a Web Technology Stack

MERN stack (**M**ongoDB, **E**xpress, **R**eact, **N**odeJS)



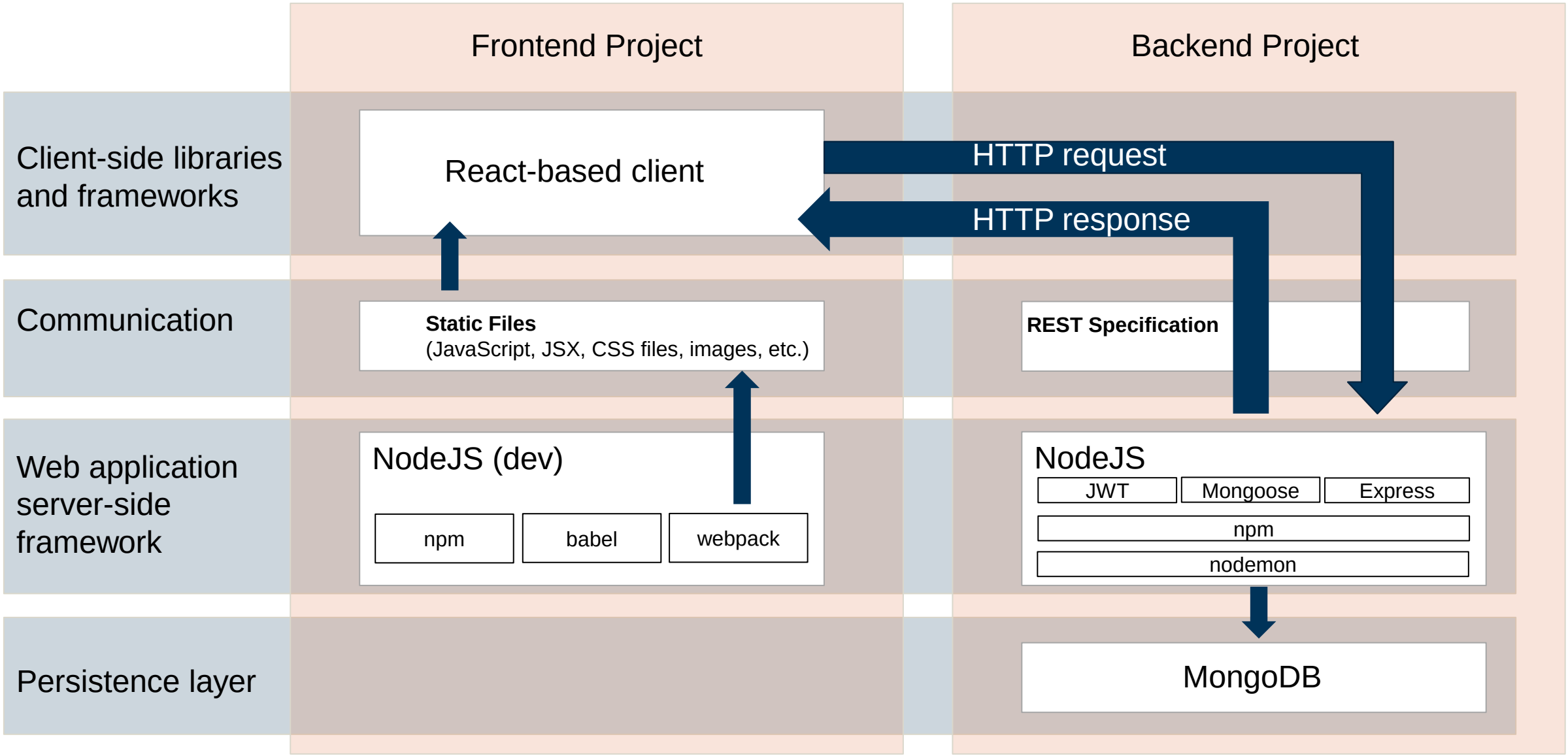
Web Development Tools

When implementing web applications, developers often face **similar issues and challenges:**

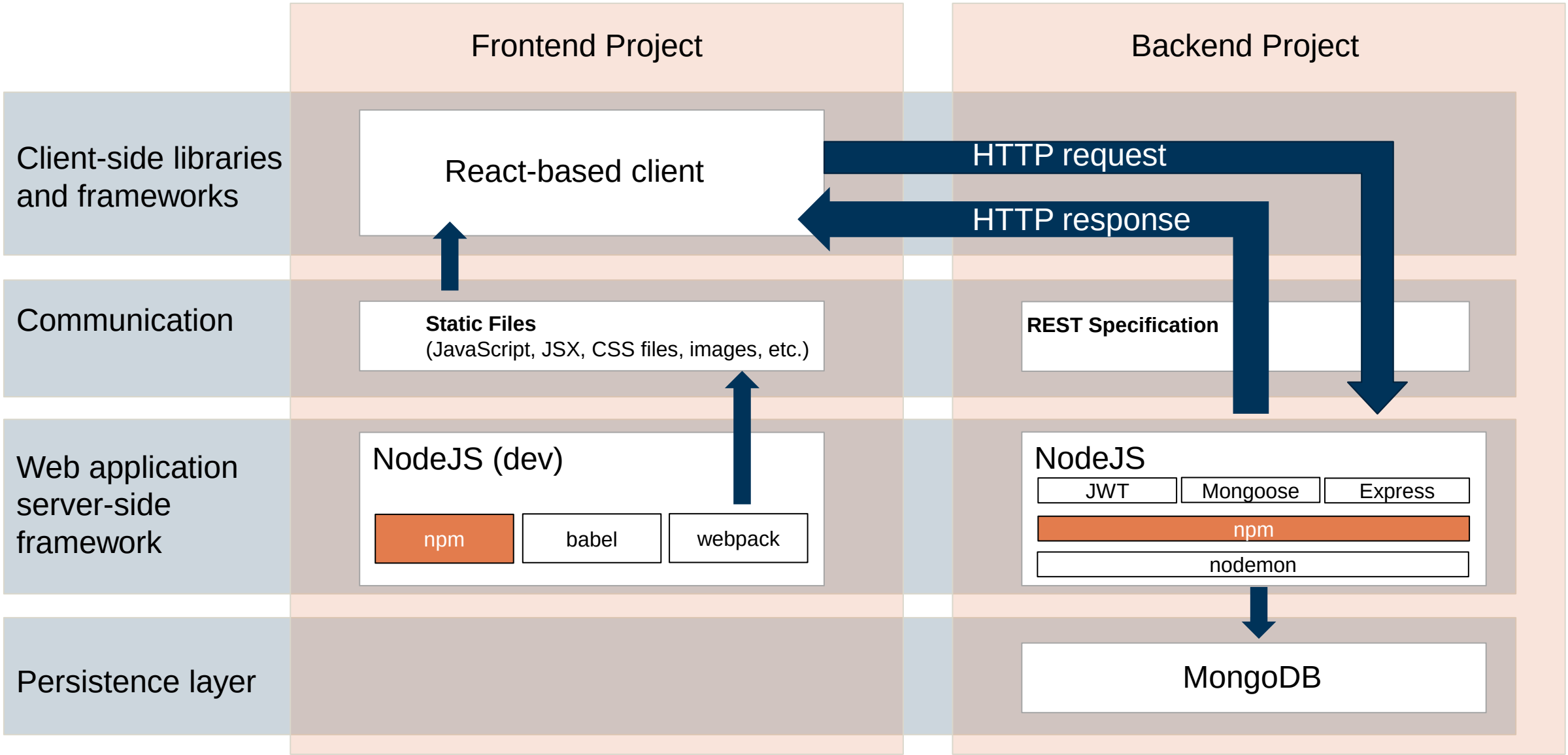
- Downloading, installing, and managing packages, frameworks, and tools for the web application server
- Downloading and managing libraries and frameworks for web clients, and ensuring the compatibility
- Preparing a basic project structure ("scaffolding")
- Transforming a component-based project structure to a deployable and executable structure
- Transcompiling a programming language (e.g., TypeScript) to an executable counterpart (e.g., JavaScript)
- Preparing and running different kinds of tests, e.g., unit tests
- Minifying or uglifying source code
- …

In previous years, a **huge ecosystem of tools** emerged which address these issues and support developers in implementing web applications.

Using MongoDB, Express (NodeJS) for the Backend Project



Node Package Manager (npm)



Node Package Manager (npm) – Package Ecosystem of NodeJS

npm is

- an online registry for open-source NodeJS modules, resources, etc.
- the official package manager for NodeJS
- the largest ecosystem of open-source libraries in the world

How to install a package:

`npm install cookie-parser (--save)`

→ adds the package "cookie-parser" to **package.json** as dependency (save flag is optional as it defaults to true)

`npm install eslint --save-dev`

→ adds the package "eslint" to **package.json** as devDependency (development dependencies are not bundled into production code)

(QoL tip: `npm install` can be written as `npm i`)

package.json

```
{
    "name": "VolunteerApp",
    "version": "1.2.0",
    "private": true,
    "scripts": {
        "start": "node./bin/www"
    },
    "dependencies": {
        "body-parser": "~1.13.2",
        "cookie-parser": "~1.3.5",
        "debug": "~2.2.0",
        "express": ">4.13.1"
    }
}
```

Application Metadata

Application Dependencies

version	Must match **version** exactly
>version	Must be greater than version
>=version	…
<version	…
<=version	…
~version	"Approximately equivalent to version"
^version	"Compatible with version"
1.2.x	1.2.0, 1.2.1, etc., but not 1.3.0
*****	Matches any version
""	(just an empty string) Same as *
etc.	check linked source for more options

Source: https://docs.npmjs.com/files/package.json

NodeJS Packages Used in Movie Database App

Dependency Name	Short Description
bcryptjs	Useful to encrypt and decrypt salted passwords
body-parser	A NodeJS middleware to parse incoming request bodies before the handlers
express	A REST enabling framework on top of NodeJS
helmet	Helpful to secure Express apps by setting various HTTP headers
jsonwebtoken	Web tokens identify users in a scalable manner and "replace" the old session id
mongoose	Connect to MongoDB

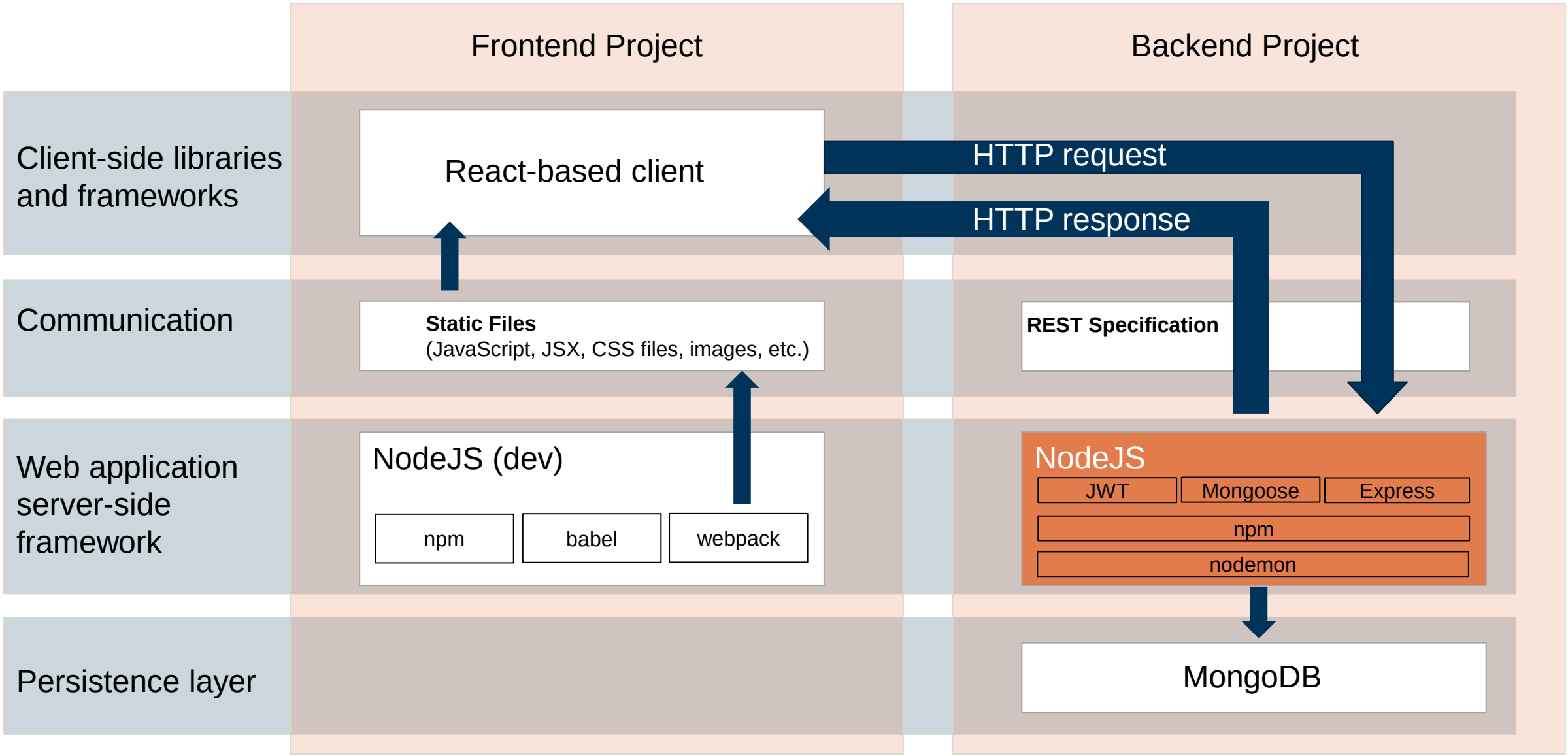
More NodeJS Packages here: https://www.npmjs.com

Outline of the Lecture

6. Building a REST-enabled Backend Service

- Target Architecture and Development Environment
- Event-driven Architecture and Asynchronous I/O Operations with NodeJS
- Creating REST Interfaces Using Express
- Using Document-oriented Database Storage: MongoDB Example
- Enabling User Authentication on the Web Service Using JSON Web Tokens (JWT)

Event-driven Architecture and Asynchronous I/O Operations with NodeJS

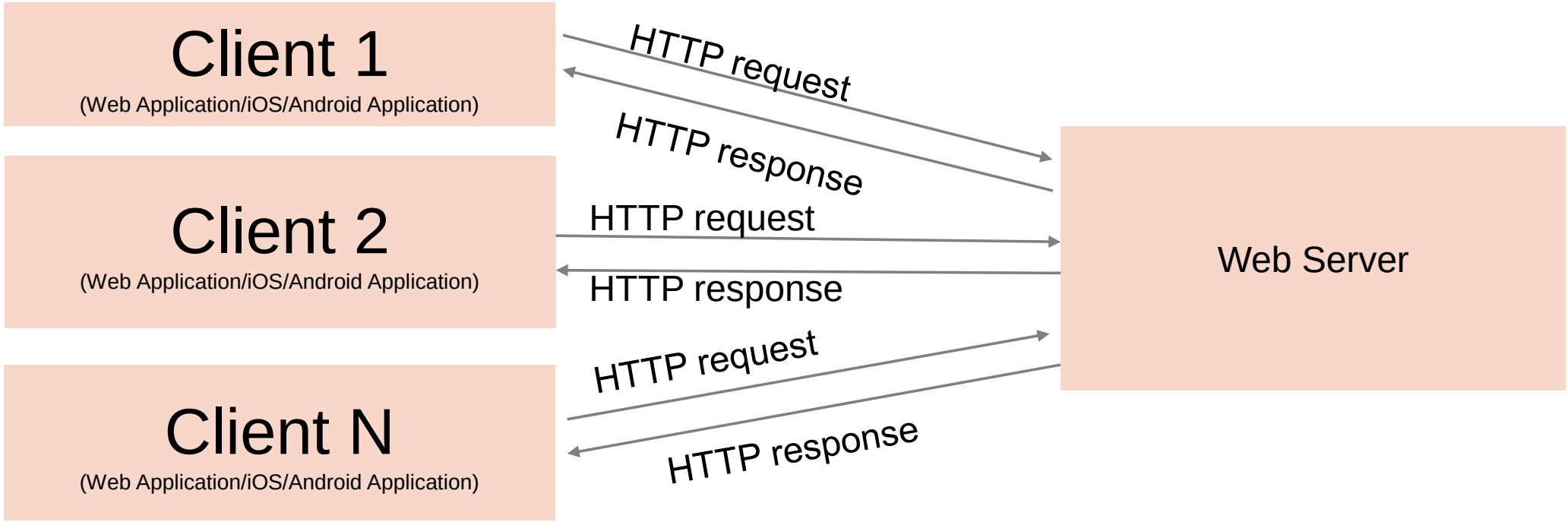


Black Box View of a REST-enabled Web Service

A web application service receives a request via HTTP with

- a URL,
- a request header,
- a request body,

and sends back an HTTP response which contains the answer encoded in **JSON.**



Handling Concurrent Requests from Multiple Clients

A) Each request is handled by a separate operating system (OS) process **(outdated)**

Example:

- CGI scripts written in Perl

Just for comparison

Pros:

- Isolation
- Security
- Multi-language support
- Availability on every OS

Cons:

- High operating cost
- High consumption of memory

Handling Concurrent Requests from Multiple Clients

B) Each request is handled by a separate thread in a single shared OS process

Example:

Multithreading in JAVA (EJB), C# (.NET)

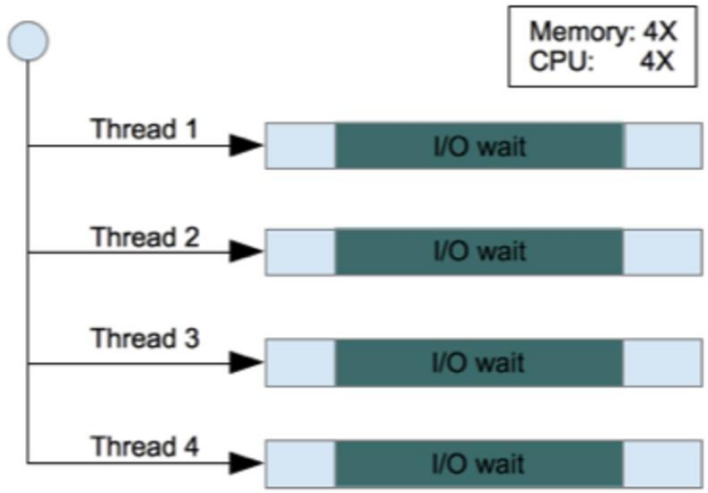
Just for comparison

Pros:

- Use of multi-core CPU under your control
- Shared main memory data structure
- One language
- No limitations on blocking IO (easy sequential programming)
- Threads can be directly mapped to DB transactions

Cons:

- Difficulty of concurrent programming
- Requires thread-safe data structures
- Need for effective thread-pooling



Handling Concurrent Requests from Multiple Clients

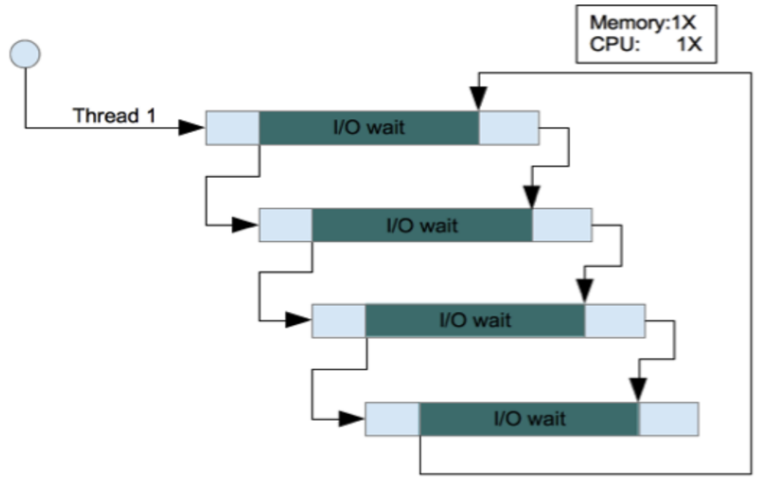
C) Each request is handled by a single thread in a single OS process

Example:
Event-driven non-blocking I/O framework (NodeJS, Netty, etc.)

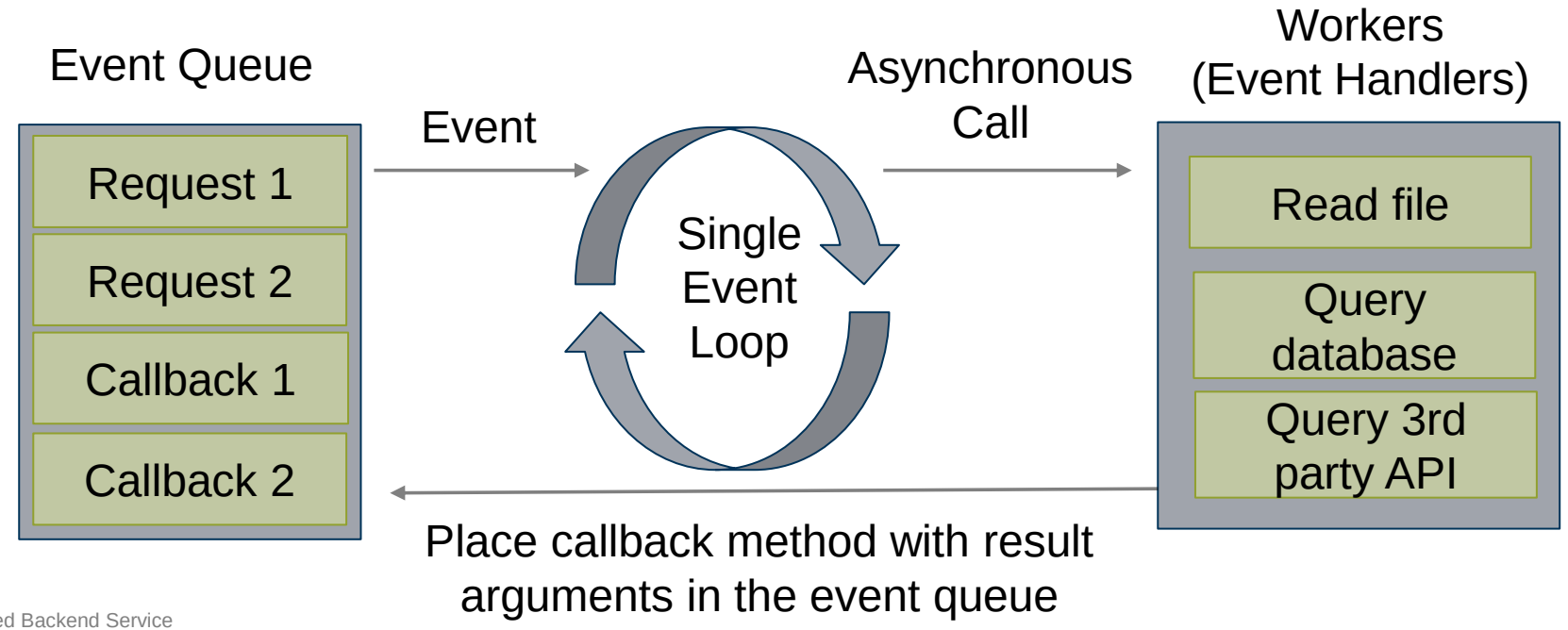
Pros:
- Lower memory consumption
- Developer does not deal with concurrency issues on the level of web application server
- Better resistance to heavy loads by design

Cons:
- Long-running computations inside single thread block the server

NodeJS – Asynchronous I/O



NodeJS example:



NodeJS

NodeJS is an **open-source**, **cross-platform** JavaScript **runtime environment** for executing JavaScript code on the **server side**.

Why developers chose NodeJS?

- Built on top V8 JavaScript engine (built and supported by Google, used inside Chrome Browser)
- Enable JavaScript on the server → no need to switch between different languages on client and server
- Cross-platform
- Popularity of JSON as a standard for exchanging content
- Huge ecosystem of libraries with active developer communities

To effectively deal with concurrent requests in NodeJS developers are forced to use:

- Non-blocking asynchronous NodeJS API methods and callbacks
- More advanced: Streams and EventEmitters (More: https://nodejs.org/en/docs/guides/backpressuring-in-streams/)

Blocking Synchronous Method vs. Non-blocking Asynchronous Method

Why callbacks?

file.md

```
File content is placed here
```

Blocking example:

Using synchronous method in the single event loop

Synchronous method:
- **Performs I/O operations (file reads, db queries, etc.)**
- **Executed inside NodeJS event loop**
- **Blocks event loop until I/O operation performed**

main.js

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

const data = fs.readFileSync('/file.md');

console.log(data);

console.log('Example with synchronous method');
```

Console output:

```
File content is placed here
Example with synchronous method
```

Non-blocking example:

Using asynchronous method with callback function

Asynchronous method:
- **Performs I/O operations (file reads, db queries, etc.)**
- **Executed outside NodeJS event loop (without blocking it)**
- **Could use multiple threads (depends on OS and realization)**
- **Gets a callback function as the last parameter**

main.js

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

fs.readFile('/file.md', (err, data) => {
    if (err) throw err;
    console.log(data);
});

console.log('Example with asynchronous method');
```

Callback Function:
- **Executed after** an error occurs or the operations is successfully completed
- **Executed inside** event loop

Console output:

```
Example with asynchronous method
File content is placed here
```

Or possibly:

```
File content is placed here
Example with asynchronous method
```

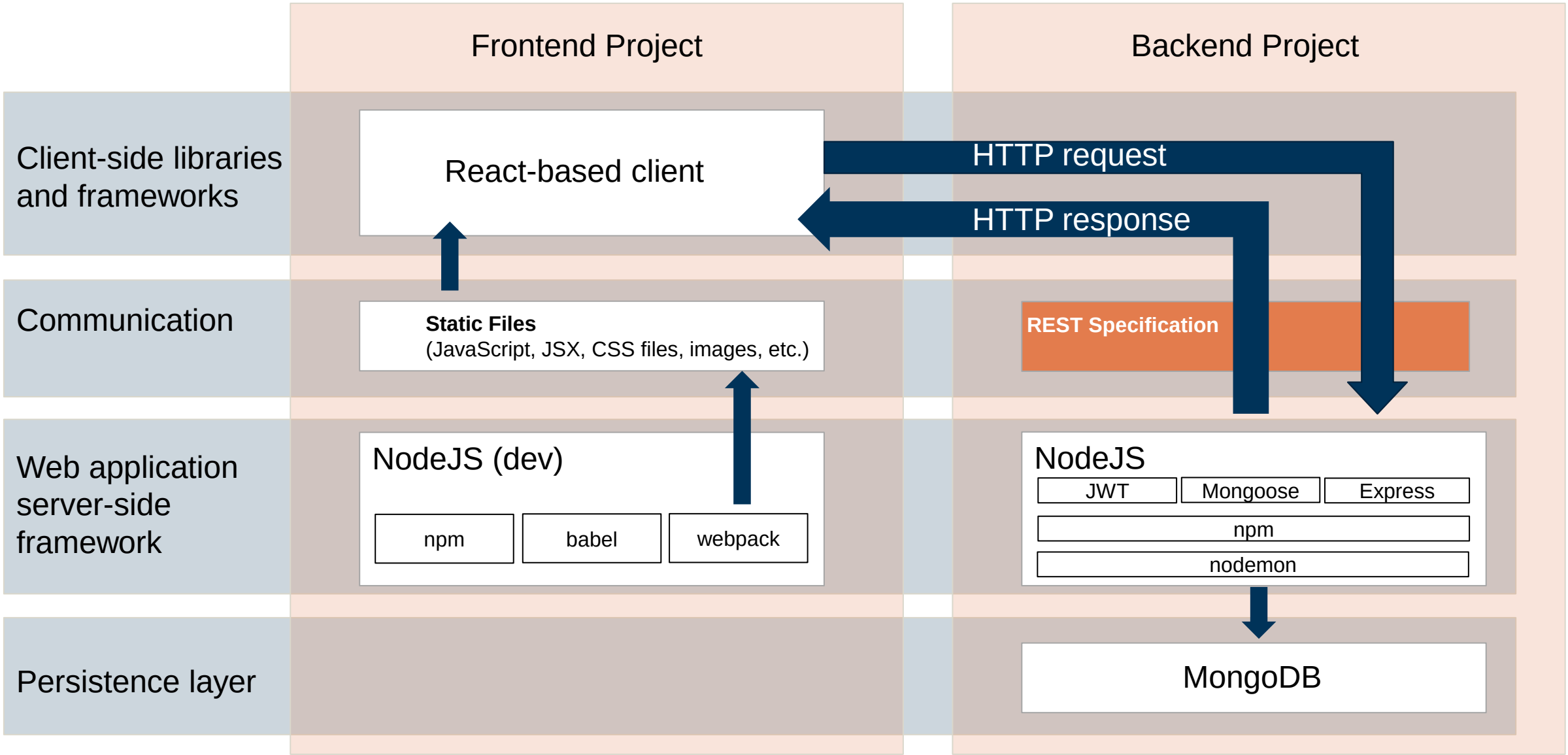
Source: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

Outline of the Lecture

6. Building a REST-enabled Backend Service

- Target Architecture and Development Environment
- Event-driven Architecture and Asynchronous I/O Operations with NodeJS
- Creating REST Interfaces Using Express
- Using Document-oriented Database Storage: MongoDB Example
- Enabling User Authentication on the Web Service Using JSON Web Tokens (JWT)

REST Specification



Representational State Transfer (REST)

- Representational State Transfer (**REST**) is a software architectural style for distributed hypermedia systems like the world wide web.
- The term has been coined by Roy Fielding in his doctoral dissertation.
- REST provides a set of **architectural constraints** that, when applied as a whole, enable:
  - scalability of component interactions,
  - generality of interfaces,
  - independent deployment of components, and
  - intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.
- The REST architectural style has been used to guide the design and development of the architecture for the modern Web.

The slides (21 – 30) summarize the concepts behind REST which we assume to be already known.

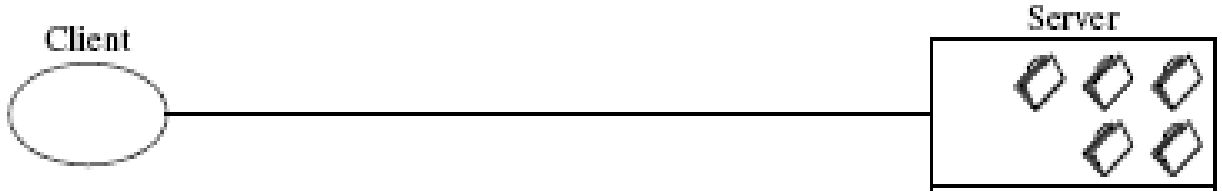
The Null Style

- Is simply an empty set of constraints.
- Describes a system in which there are no distinguished boundaries between components.
- Is the starting point for the description of REST.

- Example: Bad mainframe application

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

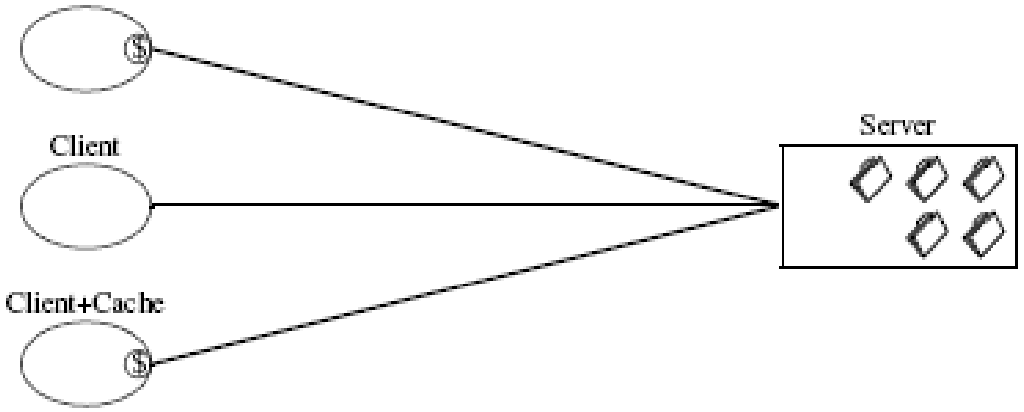
Client-Server Architecture



- Separates the user interface concerns from the data storage concerns.
- Improves the portability of the user interface across multiple platforms.
- Improves scalability by simplifying the server components.
- Allows the components to evolve independently.

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Cacheable

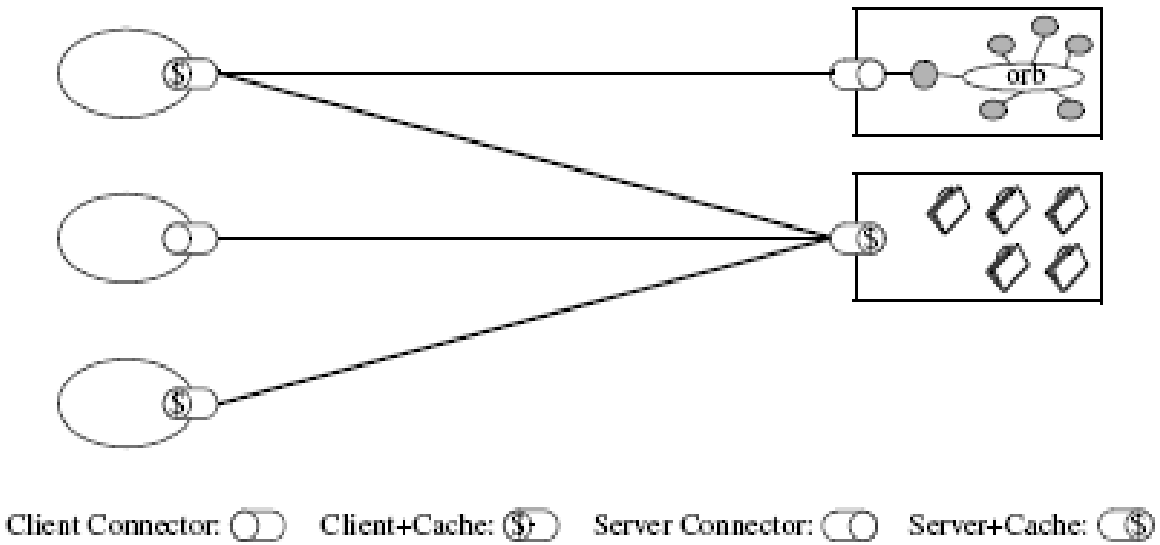


- Require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable.
- Improves efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions.

- Trade-off : a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Uniform Interfaces



- This is the central feature that distinguishes the REST architectural style from other network-based styles.
- Emphasis on a **uniform interface** between components
  - Uniform identification scheme
  - Uniform representation of information exchanged
- Implementations are **decoupled** from the services they provide.

- Trade-off: information is transferred in a standardized form rather than one which is specific to an application's needs.

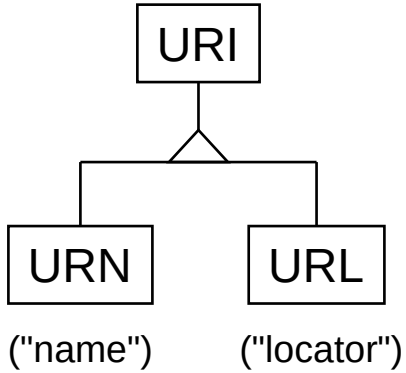
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Uniform Representation of Information Exchanged

- The type of a representation is specified in the HTTP header **`Content-Type`** of the response.
- Content types are identified using MIME (Multipurpose Internet Mail Extension) types.
- A MIME type contains a type and a subtype.
  - Types are generic and predefined, e.g., **`text`**, **`image`**, **`audio`**, **`video`**, and **`application`**.
  - Subtypes provide more information about the content, e.g., **`text/plain`**, **`text/html`**, **`image/jpeg`**, **`application/json`**.
- The Web uses standardized (often human-readable) exchange content formats, in particular HTML and XML, JSON.
- Contrast with "optimized" record layout & binary representations.

Uniform Identification Scheme

- **URI**: Uniform Resource Identifier (general term)
  - There are two mechanisms: naming and location
- **URN**: Uniform Resource Name
  - Identification of objects by name for the purpose of persistent labeling, e.g., ISBN
- **URL**: Uniform Resource Locator
  - Identification via the primary access mechanism



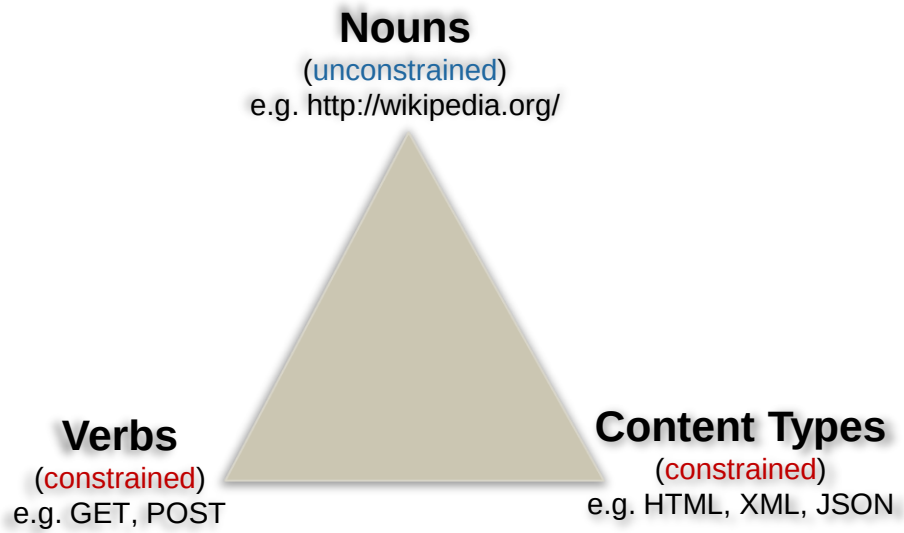
- An example of an URI:

scheme		authority	path		query		fragment
http	://	example.org	/mysite/page	?	name=cat	#	whiskers

- A URI points to a hierarchical space reading from left to right; each block that follows is a branch from the previous block.
- URIs are not being used uniformly on the Web, but: when a URI has been used to identify a resource, it should continue to be used to identify the same resource.
- See: "**Cool URIs don't change**" by Tim Berners-Lee [http://www.w3.org/Provider/Style/URI]

REST Triangle

- The main problem domains identified in REST are the **nouns**, the **verbs**, and the **content-type** spaces.
- The *things that exist*, the *things you can do to them*, and the *information you can transfer* as part of any particular operation



- REST requires a **standardized** set of state transfer operations.

Mapping HTTP With REST Methods

Minimum methods:

- **GET** is the HTTP equivalent of COPY
  - Transfers a representation from resource to client.
- **PUT** is the HTTP equivalent of PASTE OVER
  - Transfers state from a client to a resource.
  - GET and PUT are fine for transferring state of existing resources.
- **POST** is the PASTE AFTER verb
  - Don't overwrite what you currently have, add to it.
  - Create a resource.
  - Add to a resource.
- **DELETE** is the HTTP equivalent of CUT
  - Requests the resource state being destroyed.

Example:
Navigating to www.tum.de/studium results in the following GET request:

```
GET /studium HTTP/1.1
Host: www.tum.de
...
```

Further HTTP verbs/methods: HEAD, PATCH, TRACE, OPTIONS, CONNECT

Source: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

Web APIs do not provide presentation elements but services to be consumed by application code

- A Web API consists of a defined set of HTTP Request messages.
- For each request, the Web API specifies the structure of response messages.
- A response message is typically expressed in JSON or XML.

- **JSON** (JavaScript Object Notation) is a lightweight text-based open standard designed for human-readable data interchange.
- It is derived from the object literal syntax of JavaScript programming language.

- Two equivalent responses in different formats:

XML

```xml
<lecture>
  <title>SEBA Master</title>
  <chapters>
    <chapter>
      <number>1</number>
      <title>Web Site Genres</title>
    </chapter>
    <chapter>
      <number>2</number>
      <title>Web Site Design Process</title>
    </chapter>
    <chapter>
      <number>3</number>
      <title>Web Design Patterns</title>
    </chapter>
  </chapters>
</lecture>
```

JSON

```json
{
  "title": "SEBA Master",
  "chapters": [
    {
      "number": 1,
      "title": "Web Site Genres"
    },
    {
      "number": 2,
      "title": "Web Site Design Process"
    },
    {
      "number": 3,
      "title": "Web Design Patterns"
    }
  ]
}
```

REST API in Practice: The Twitter API (1)

Some of the available methods:

statuses

```
GET statuses/mentions_timeline
GET statuses/home-timeline
GET statuses/retweets_of_me
GET statuses/retweets/:id
GET statuses/show/:id
GET statuses/destroy/:id
GET statuses/update
GET statuses/retweet:id
GET statuses/update_with_media
GET statuses/oembed
GET statuses/retweeters/ids
GET statuses/lookup
```

media

```
POST media/upload
```

direct_messages

```
GET direct_messages
POST direct_messages/destroy
POST direct_messages/new
```

friendships

```
GET friendships/no_retweets/ids
GET friendships/incoming
GET friendships/outgoing
POST friendships/create
POST friendships/destroy
POST friendships/update
GET friendships/show
GET friendships/lookup
```

friends

```
GET friends/ids
GET friends/list
```

followers

```
GET friends/ids
GET followers/list
```

and many more…

Fielding would not call that REST:
Using POST for a deletion or GET for an update is bad practice!

Source: https://developer.twitter.com/en/docs/api-reference-index

REST API in Practice: The Twitter API (2)

Example: Retrieving the IDs of all retweeters of a certain status

Request:

```
GET
https://api.twitter.com/1.1/statuses/retweeters/ids.json?id=327473909412814850&count=100&stringify_ids=true
```

Answer:


```
{
    "previous_cursor": 0,
    "ids": [ "1382021622", "931150754", "1364953914", "92313481", "1398853771"],
    "previous_cursor_str": "0",
    "next_cursor": 0,
    "next_cursor_str": "0"
}
```


A list of free APIs (for use in software and web development): https://github.com/public-apis/public-apis

REST API in Practice: API Specification

- There are multiple frameworks and languages for specifying (REST) Web APIs
  - Swagger: http://swagger.io
  - RAML: http://raml.org
  - …

- REST API specification through JSON (or YAML) document
  - **General information** about the REST API
    - How to connect to the REST API?
    - What is the data format for the REST API (e.g., JSON, XML)?
  - **Definition of resource types**
    - Which resources are available through the REST API?
    - Which attributes of which types (e.g., integer) do the resources have?
  - **Definitions of operations**
    - Which operations are available?
    - Which REST verb to use for which URL?
    - Which parameters are allowed and/or required?
    - How does the response look like?
    - How does errors get reported?

REST API in Practice: Exemplary OpenAPI Specification With Swagger (1)

General information / metadata

Schema definitions

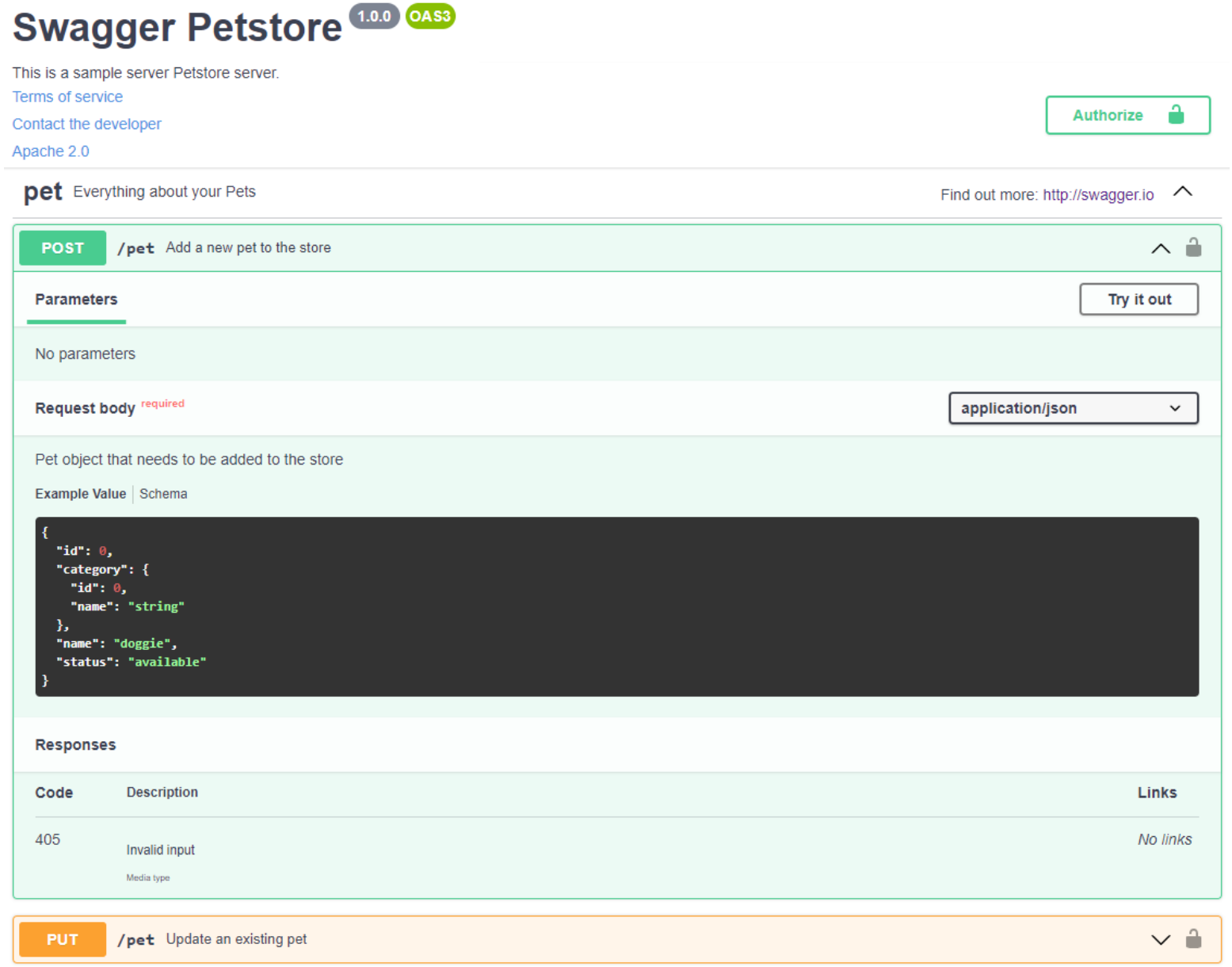
Available paths and operations

```
{
  "openapi": "3.0.2",
  "info": {
    "title": "Swagger Petstore",
    "description": "This is a sample server Petstore server.",
    "termsOfService": "http://swagger.io/terms/",
    "contact": {
      "email": "apiteam@swagger.io"
    },
    "license": {
      "name": "Apache 2.0",
      "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
    },
    "version": "1.0.0"
  },
  "components": {
    "schemas": {
      "Pet": {
        "required": [ "name" ],
        "type": "object",
        "properties": {
          "id": { "type": "integer", "format": "int64" },
          "category": { "$ref": "#/components/schemas/Category" },
          "name": { "type": "string", "example": "doggie" },
          "status": {
            "type": "string",
            "description": "pet status in the store",
            "enum": [ "available", "pending", "sold" ]
          }
        },
      },
      ... more schema definitions ...
    }
  },
```

```
  "paths": {
    "/pet": {
      "post": {
        "tags": [ "pet" ],
        "summary": "Add a new pet to the store",
        "operationId": "addPet",
        "requestBody": {
          "description": "Pet object that needs to be added to the store",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/Pet"
              }
            },
            "application/xml": {
              "schema": {
                "$ref": "#/components/schemas/Pet"
              }
            }
          },
          "required": true
        },
        "responses": {
          "405": {
            "description": "Invalid input",
            "content": {}
          }
        },
        "security": [ { "petstore_auth": [ "write:pets", "read:pets" ] } ]
      },
      "put": { ... }
    },
    "/pet/findByStatus": { ... }
  }
  ...
}
```

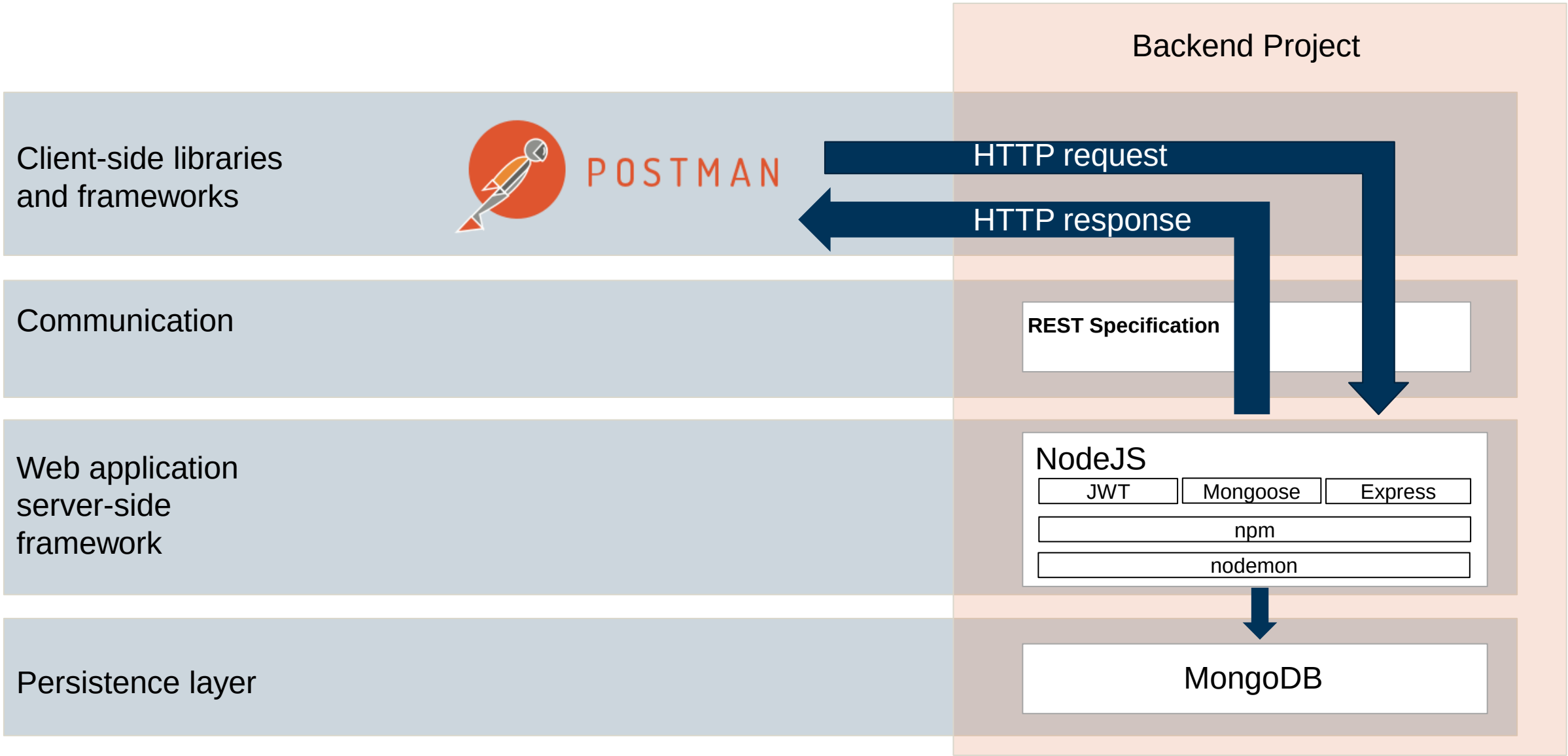
https://swagger.io/specification/

REST API in Practice: Exemplary OpenAPI Specification With Swagger (2)

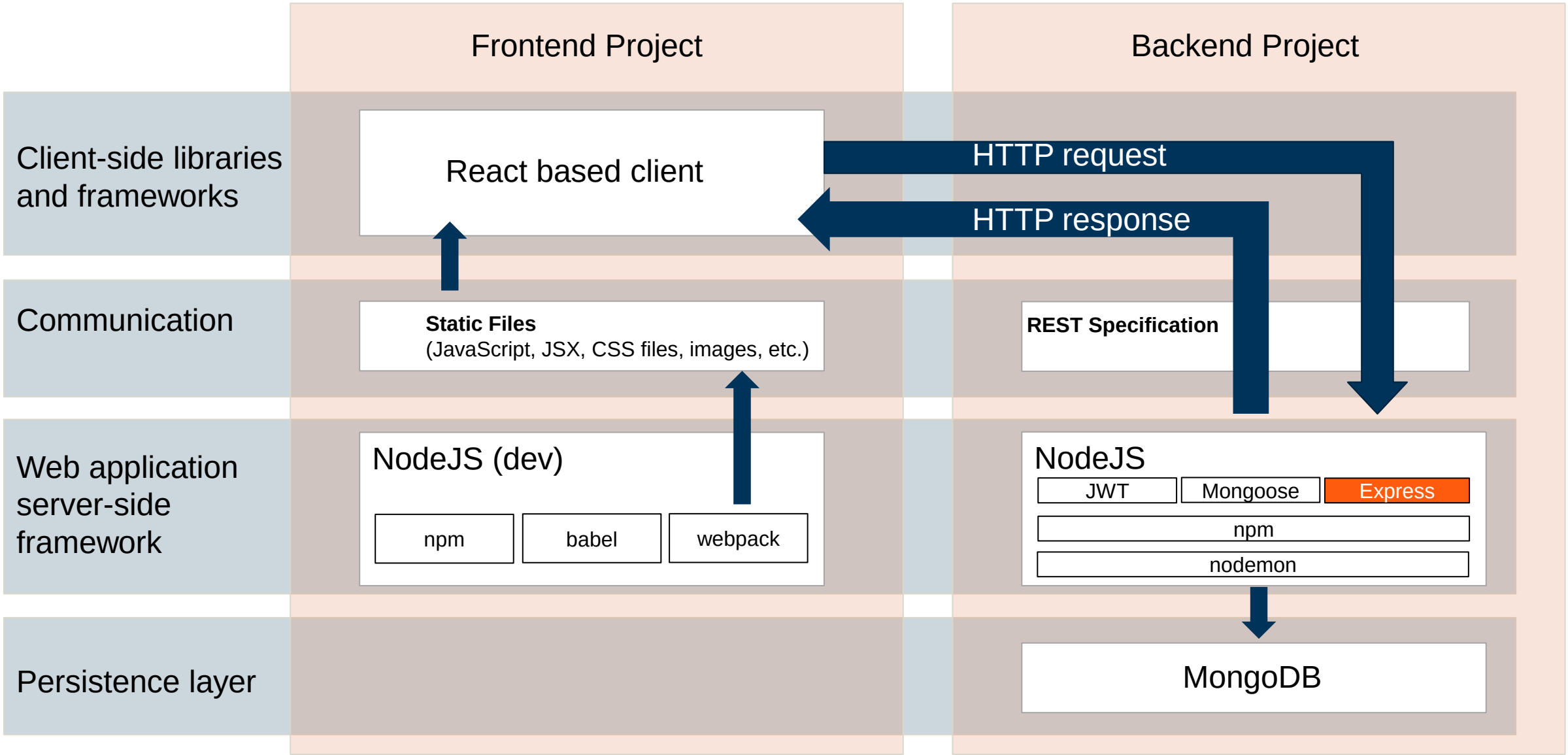


https://editor.swagger.io/

Postman Allows to Interactively Test Your REST API



Creating REST Interfaces With Express



Express: A Web Application Framework

Server-side web frameworks (a.k.a. "web application frameworks") are software frameworks that make it easier to write, maintain and scale web applications. They provide tools and libraries that simplify common web development tasks, including **routing URLs to appropriate handlers, interacting with databases, supporting sessions and user authorization, formatting output (e.g. HTML, JSON, XML), and improving security against web attacks.**

https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks

Express is a web application framework on top of Node.js

Express provides a **thin layer** of **fundamental web application features**, without obfuscating Node.js features. →

In particular, it is a set of higher level asynchronous Node.js methods that were implemented by experienced developers taking into account:

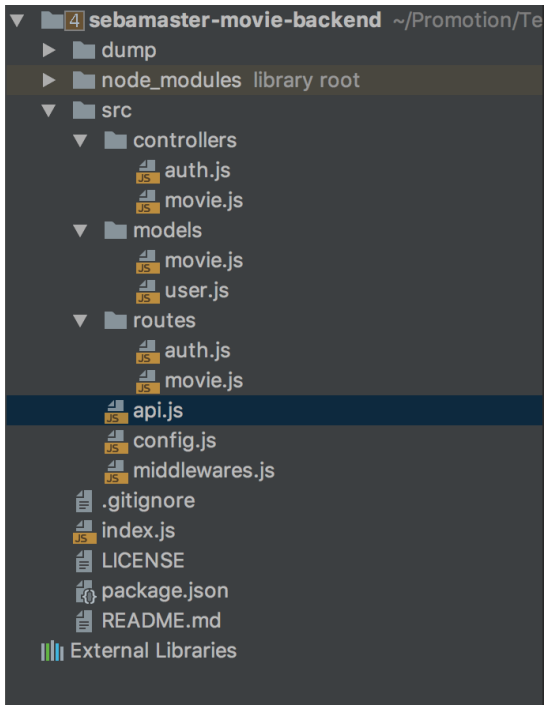
- Security
- Error handling
- Routing
- Middleware modules (request parsers, cors, authentication module (passport), compressors) (more)
- Template engines (avoided in our example architecture, REST interfaces instead)

Express is "unopinionated" → It's up to the developer to use certain middleware modules or templates engines (developers rely on their own decisions and experiences of others)

Similar frameworks: Koa, Hapi, Restify

Frameworks based on top of Express (usually opinionated): LoopBack, Sails, Locomotive

Express: Typical Project Structure, App Initialization and Middleware



api.js example

```
"use strict";

const express    = require('express');
const bodyParser = require('body-parser');
const helmet     = require('helmet');

const middlewares = require('./middlewares');

const auth  = require('./routes/auth');
const movie = require('./routes/movie');

const api = express();


// Adding Basic Middlewares
api.use(helmet());
api.use(bodyParser.json());
api.use(bodyParser.urlencoded({ extended: false }));
api.use(middlewares.allowCrossDomain);


// Basic route
api.get('/', (req, res) => {
    res.json({
        name: 'SEBA Master Movie Backend'
    });
});

// API routes
api.use('/auth'   , auth);
api.use('/movies', movie);



module.exports = api;
```

index.js example

```
'use strict';

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

const api        = require('./src/api');
const config     = require('./src/config');


// Set the port to the API.
api.set('port', config.port);

//Create a http server based on Express
const server = http.createServer(api);


//Connect to the MongoDB database; then start the server
mongoose
    .connect(config.mongoURI)
    .then(() => server.listen(config.port))
    .catch(err => {
        console.log('Error connecting to the database', err.message);
        process.exit(err.statusCode);
    });


server.on('listening', () => {
    console.log(`API is running in port ${config.port}`);
});

server.on('error', (err) => {
    console.log('Error in the server', err.message);
    process.exit(err.statusCode);
});
```

Express: Defining REST API Resources (Routing)

movie.js example

```
"use strict";

const express  = require('express');
const router   = express.Router();

const middlewares    = require('../middlewares');
const MovieController = require('../controllers/movie');


router.get('/', MovieController.list); // List all movies
router.post('/', middlewares.checkAuthentication, MovieController.create); // Create a new movie
router.get('/:id', MovieController.read); // Read a movie by Id
router.put('/:id', middlewares.checkAuthentication, MovieController.update); // Update a movie by Id
router.delete('/:id', middlewares.checkAuthentication, MovieController.remove); // Delete a movie by Id


module.exports = router;
```

Express: Controllers

movie.js example

```
"use strict";

const MovieModel = require('../models/movie');


const create = (req, res) => {
    if (Object.keys(req.body).length === 0) return res.status(400).json({
        error: 'Bad Request',
        message: 'The request body is empty'
    });

    MovieModel.create(req.body)
        .then(movie => res.status(201).json(movie))
        .catch(error => res.status(500).json({
            error: 'Internal server error',
            message: error.message
        }));
};

const read   = (req, res) => {
    MovieModel.findById(req.params.id).exec()
        .then(movie => {

            if (!movie) return res.status(404).json({
                error: 'Not Found',
                message: `Movie not found`
            });

            res.status(200).json(movie)

        })
        .catch(error => res.status(500).json({
            error: 'Internal Server Error',
            message: error.message
        }));

};

const update = (req, res) => {
    if (Object.keys(req.body).length === 0) return res.status(400).json({
        error: 'Bad Request',
        message: 'The request body is empty'
    });

    MovieModel.findByIdAndUpdate(req.params.id,req.body,{ new: true, runValidators: true}).exec()
        .then(movie => res.status(200).json(movie))
        .catch(error => res.status(500).json({
            error: 'Internal server error',
            message: error.message
        }));
};
```

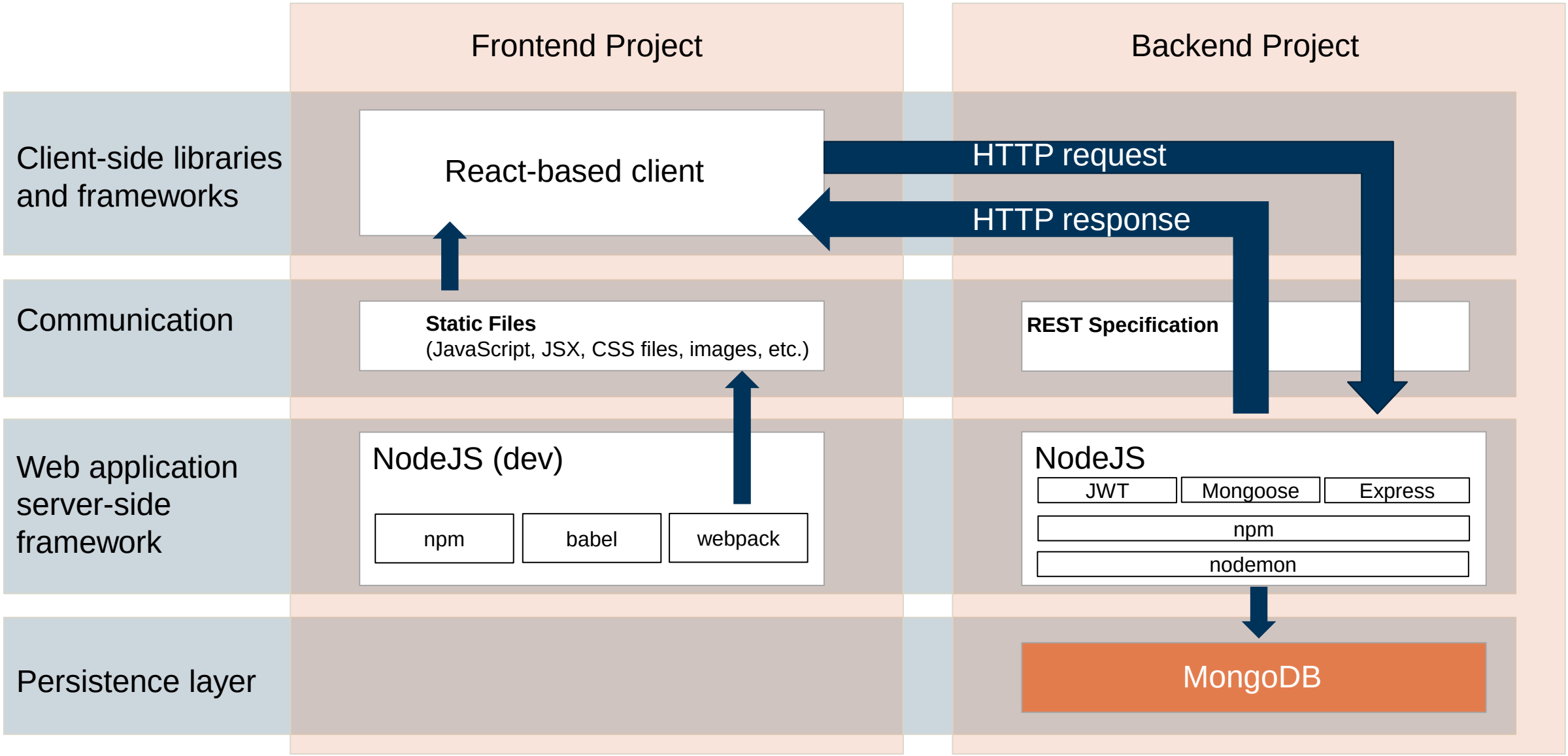
```

const remove = (req, res) => {
    MovieModel.findByIdAndRemove(req.params.id).exec()
        .then(() => res.status(200).json({message: `Movie with id${req.params.id} was deleted`}))
        .catch(error => res.status(500).json({
            error: 'Internal server error',
            message: error.message
        }));
};

const list  = (req, res) => {
    MovieModel.find({}).exec()
        .then(movies => res.status(200).json(movies))
        .catch(error => res.status(500).json({
            error: 'Internal server error',
            message: error.message
        }));
};



module.exports = {
    create,
    read,
    update,
    remove,
    list
};
```

Outline of the Lecture

6. Building a REST-enabled Backend Service

- Target Architecture and Development Environment
- Event-driven Architecture and Asynchronous I/O Operations with NodeJS
- Creating REST Interfaces Using Express
- Using Document-oriented Database Storage: MongoDB Example
- Enabling User Authentication on the Web Service Using JSON Web Tokens (JWT)

Using Document-oriented Database Storage: MongoDB Example



NoSQL Databases in Web Application Development

Reasons for popularity in web development:

- Flexible data model (database schema can evolve with business requirements)
- Complex data types
- Scaling
- API easy to work with (no need for object relational mapping)

MySQL	MongoDB

`INSERT INTO users (user_id, age, status)`
`VALUES ('bcd001', 45, 'A')`
	`db.users.insert({`
	`  user_id: 'bcd001',`
	`  age: 45,`
	`  status: 'A'`
	`})`

`SELECT * FROM users`	`db.users.find()`

`UPDATE users SET status = 'C'`
`WHERE age > 25`
	`db.users.update(`
	`  { age: { $gt: 25 } },`
	`  { $set: { status: 'C' } },`
	`  { multi: true }`
	`)`

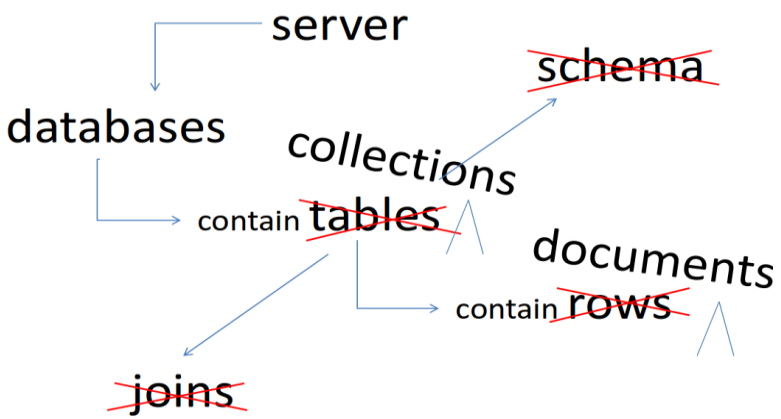
When relational databases are still better to use?

- In applications that require complex, multi-row transactions, e.g., travel reservation system or core banking application (both rely on complex transactions).
- In legacy applications, where all business logic is built around a relational data model and SQL.

Document-oriented Databases

MongoDB - Terminology

RDBMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key: _id)
Partition	Shard
Partition Key	Shard Key
Database server and client	
mysqld	mongod
mysql	mongo

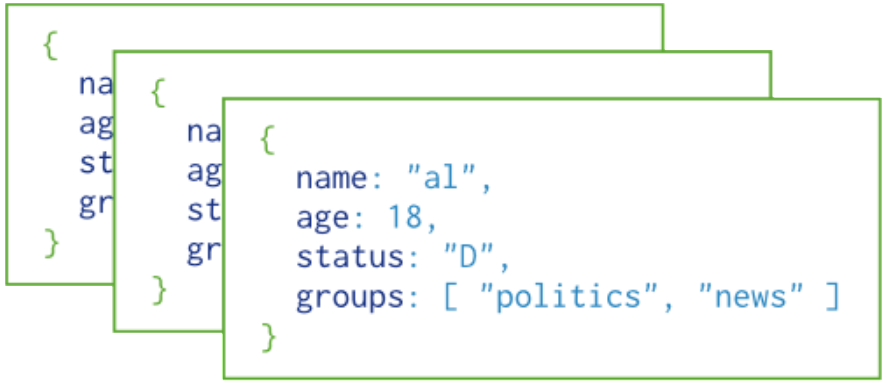


- Collections do not enforce a schema.
- Collections can be created on demand.
- A document is a set of key-value pairs and can have a dynamic schema.
- Typically, all documents within a collection are of similar or related purpose.

Document-oriented Databases

Sample Document and Collection

```
{
  name: "sue",                                  <-- field: value
  age: 26,                                      <-- field: value
  status: "A",                                  <-- field: value
  groups: [ "news", "sports" ]                  <-- field: value
}
```

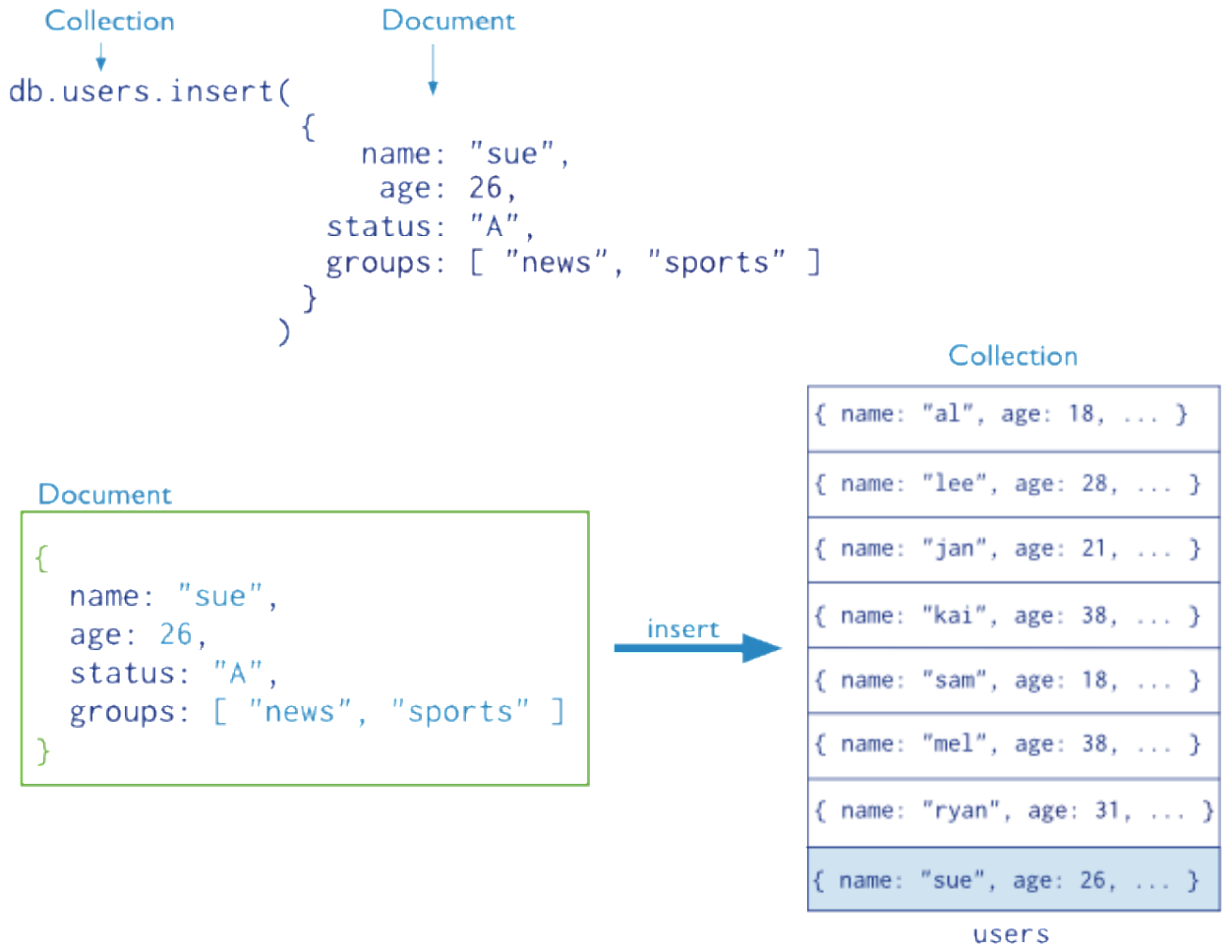


String	Integer
Boolean	Double
Arrays	Timestamp
Object	Null
Symbol	Date
Object ID	Binary Data
JS Code	Reg Expressions

[Document databases, MongoDB]

Document-oriented Databases

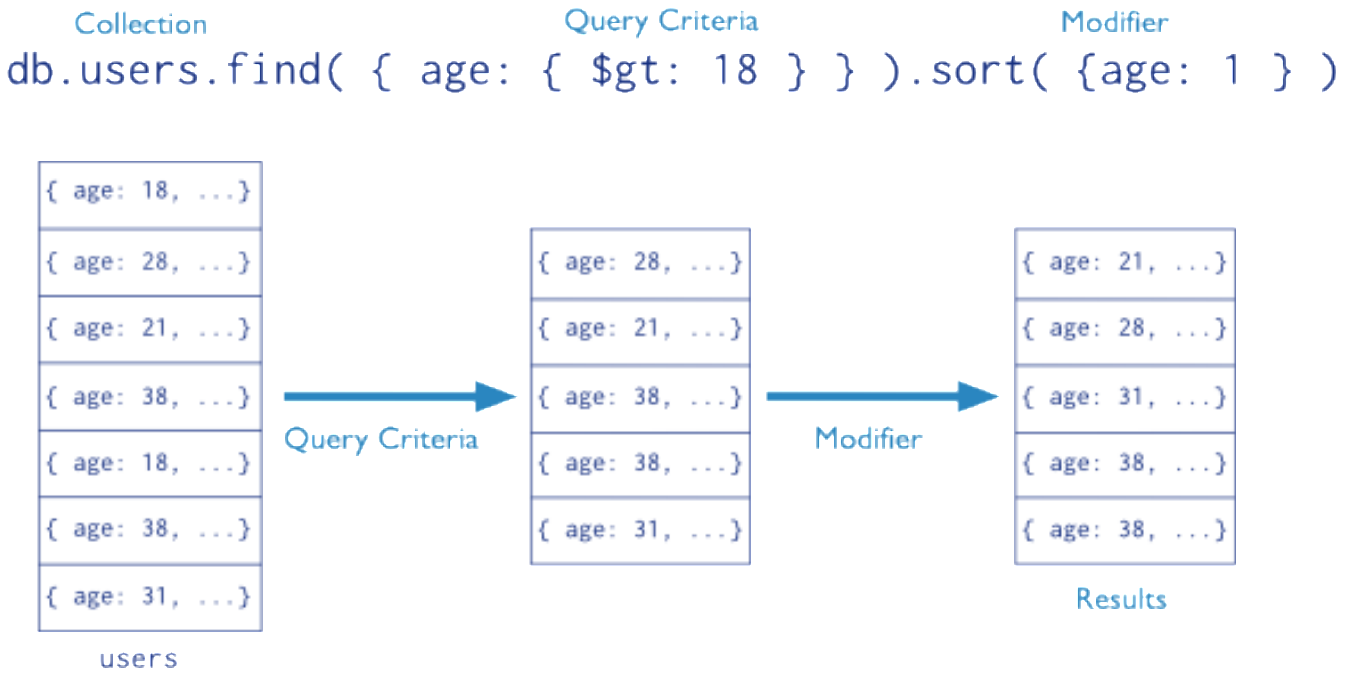
MongoDB – Inserting Document Into Collection



[Document databases, MongoDB]

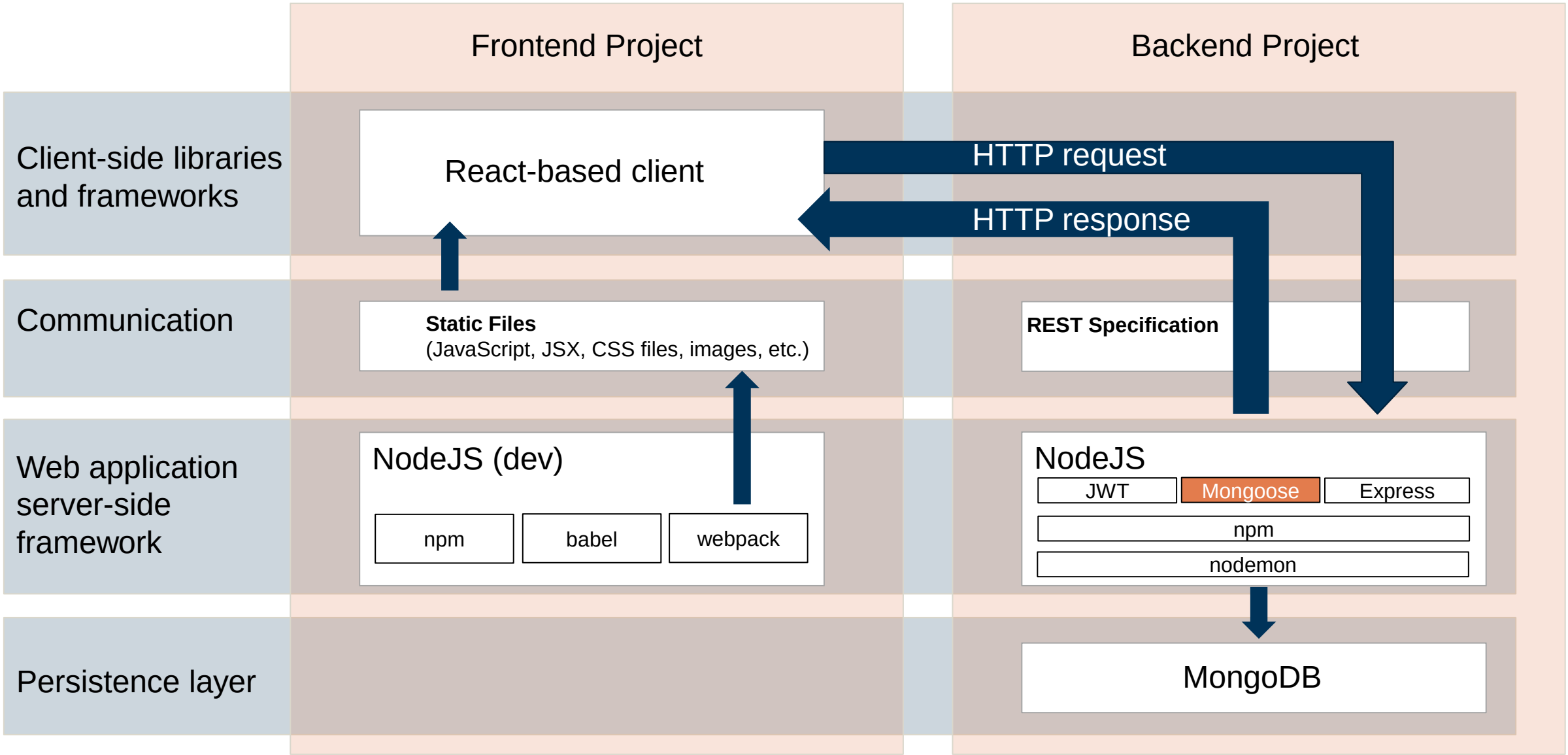
Document-oriented Databases

MongoDB – Querying Data



[Document databases, MongoDB]

Mongoose Object Document Mapper for MongoDB Concepts to JS Objects



Connecting to the Database

There are two main ways to connect to MongoDB from a NodeJS app:

- MongoDB Node.js native driver (https://github.com/mongodb/node-mongodb-native/)
  - Pros:
    + Preserves the schemaless paradigm, no modelling required from the beginning
    + Supported by MongoDB developers
  - Cons:
    - Type casting & data validation have to be implemented by the developers on their own inside NodeJS app (more difficult for beginners)

- Mongoose (http://mongoosejs.com/)
  - Pros:
    + Data validation, type casting, business logic boilerplates are provided to the developers
    + By design introduces the concept of data models in your NodeJS application providing separation of concerns
  - Cons:
    - Data models have to be introduced from the beginning (contradicts the schemaless paradigm)

Defining a Data Schema With Mongoose

movie.js: schema definition

```
"use strict";

const mongoose = require('mongoose');

// Define the movie schema

const MovieSchema  = new mongoose.Schema({
    title: {
        type: String,
        required: true
    },
    synopsis: String,
    runtime: Number,
    mpaa_rating: String,
    year: {
        type: Number,
        required: true
    },
    posters: {
            thumbnail: String,
            profile: String,
            detailed: String,
            original: String
        }
});

MovieSchema.set('versionKey', false);
MovieSchema.set('timestamps', true);

// Export the Movie model
module.exports = mongoose.model('Movie', MovieSchema);
```

user.js: schema definition

```
"use strict";

const mongoose = require("mongoose");

// Define the user schema
const UserSchema = new mongoose.Schema({
    username: {
        type: String,
        required: true,
        unique: true
    },
    password: {
        type: String,
        required: true
    }
});

UserSchema.set("versionKey", false);

// Export the User model
module.exports = mongoose.model( name: "User", UserSchema);
```

mongoose documentation: http://mongoosejs.com/docs/guide.html

Resulting JS objects

Movie
_id: ObjectId title: String synopsis: String runtime: Number mpaa_rating: String year: Number posters: { thumbnail: String, … }
find (object conditions, function cb) : Query findById (number id, function cb) : Query update (object conditions, object updates) : Query ... [many more] ...

User
_id: ObjectId username: String password: String
find (object conditions, function cb) : Query ... [many more] ...

Utilizing Resulting JS objects

movie.js

```
"use strict";

const MovieModel = require('../models/movie');
```

```
const update = (req, res) => {
    if (Object.keys(req.body).length === 0)
    {
        return res.status(400).json({
            error: 'Bad Request',
            message: 'The request body is empty'
        });
    }

    MovieModel.findByIdAndUpdate(req.params.id,req.body,{
        new: true,
        runValidators: true}).exec()
        .then(movie => res.status(200).json(movie))
        .catch(error => res.status(500).json({
            error: 'Internal server error',
            message: error.message
        }));
};
```

Example:
findByIdAndUpdate() – predefined asynchronous method defined for mongoose Schema object, that allows to update the document in the collection if the documents exists, otherwise error occurs.

auth.js

```
"use strict";

const jwt        = require('jsonwebtoken');
const bcrypt     = require('bcryptjs');

const config     = require('../config');
const UserModel  = require('../models/user');


const login = (req,res) => {
    if (!Object.prototype.hasOwnProperty.call(req.body, 'password')) return res.status(400).json({
        error: 'Bad Request',
        message: 'The request body must contain a password property'
    });

    if (!Object.prototype.hasOwnProperty.call(req.body, 'username')) return res.status(400).json({
        error: 'Bad Request',
        message: 'The request body must contain a username property'
    });

    UserModel.findOne({username: req.body.username}).exec()
        .then(user => {

            // check if the password is valid
            const isPasswordValid = bcrypt.compareSync(req.body.password, user.password);
            if (!isPasswordValid) return res.status(401).send({token: null });

            // if user is found and password is valid
            // create a token
            const token = jwt.sign({ id: user._id, username: user.username }, config.JwtSecret, {
                expiresIn: 86400 // expires in 24 hours
            });

            res.status(200).json({token: token});

        })
        .catch(error => res.status(404).json({
            error: 'User Not Found',
            message: error.message
        }));

};
```

Outline of the Lecture

6. Building a REST-enabled Backend Service

- Target Architecture and Development Environment
- Event-driven Architecture and Asynchronous I/O Operations with NodeJS
- Creating REST Interfaces Using Express
- Using Document-oriented Database Storage: MongoDB Example
- Enabling User Authentication on the Web Service Using JSON Web Tokens (JWT)

Access Control: User/Client Authentication and Authorization

Access controls gives web service owners the ability to control, restrict, monitor, and protect resource availability, integrity and confidentiality.

Access Control Challenges

- Various types of users need different levels of access (content readers, editors, platform administrators, etc.)
- Resources have different classification levels (Confidential, internal use only, private, public, etc.)
- Diverse identity data must be kept on different types of users (Credentials, personal data, contact information, work-related data, digital certificates, cognitive passwords, etc.)
- The corporate environment is continually changing (Business environment needs, resource access needs, employee roles, actual employees, etc.)

Access Control Criteria

- Roles
- Groups
- Location
- Time
- Transaction Type

Authentication: Verification of the identity of a user or a client

Authorization: Verification of a user's or client's permissions to access a resource

More here: https://en.wikibooks.org/wiki/Fundamentals_of_Information_Systems_Security/Access_Control_Systems

Methods for User and Client Authentication

Manual implementation of authorization is time consuming and error prone.

The following has to be taken into account:

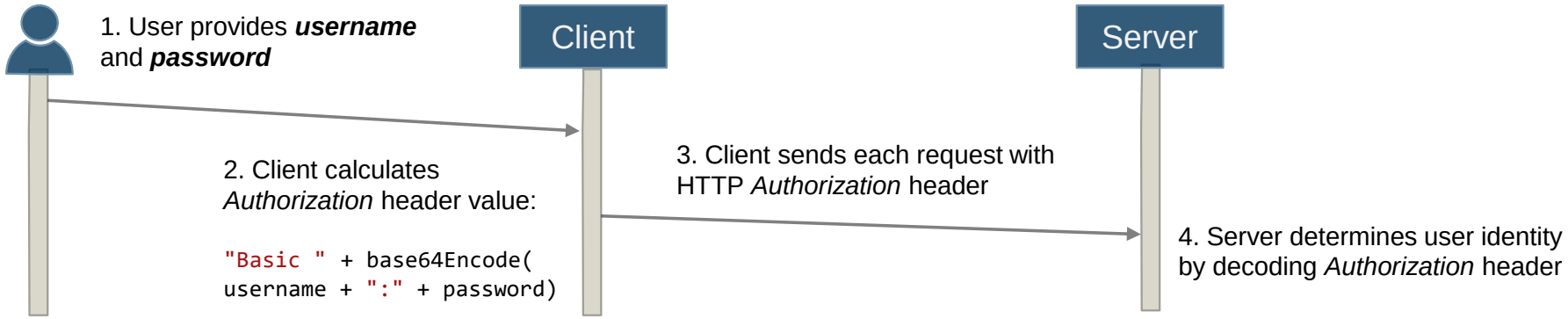
- Simple log in/sign up flow
- Encryption and secure data transfer
- Password reset
- Email verification
- Sustainability against different brute force attacks
- …

There are different types of **HTTP/HTTPS-based methods** for user and client authentication methods addressing above challenges:

- Basic Authentication
- Session-based Authentication
- JSON Web Tokens (JWT)
- OpenID
- oAuth

All of them supported by specifically tailored modules/packages/libraries in various languages (implemented and maintained by communities of developers)

Basic Authentication

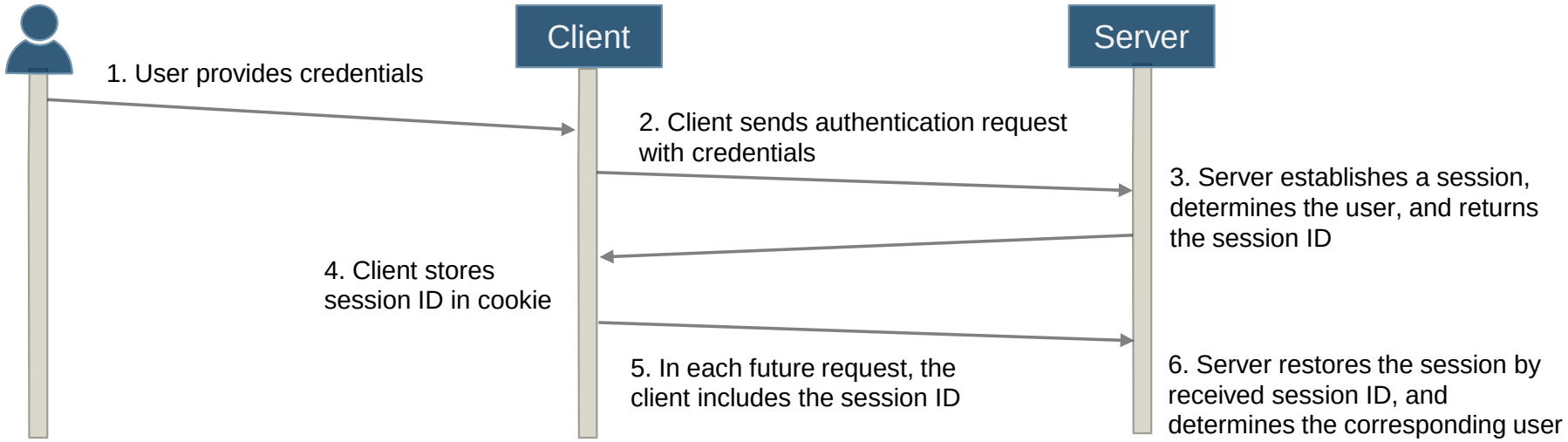


\+ Simple, improves interoperability
- Any kind of client can use basic authentication.

\+ Stateless
- Each request contains everything required to determine the user identity.

– Basically, no security measures
- Although Base64-encoded, the password is sent in plain text. Therefore, it should only be used in conjunction with SSL / HTTPS.
- On the client, the password is cached or permanently stored in plain text for future requests to the server.

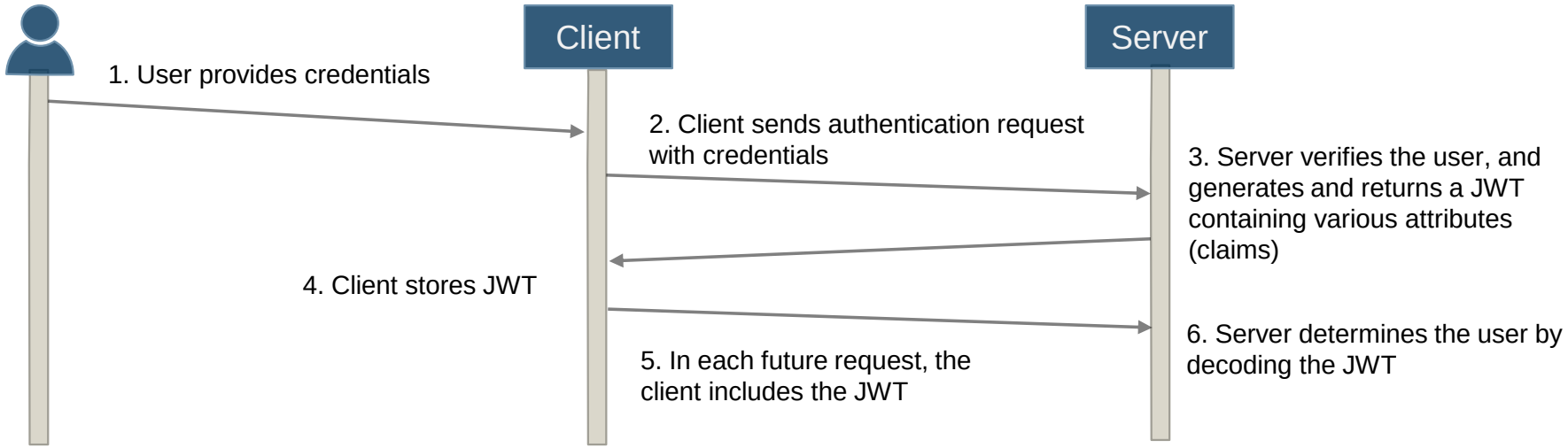
Session-based Authentication (With Cookies)



\+ More secure than Basic Authentication
- Apart from the initial exchange of credentials, the password is not included in the requests anymore.
- Password has not to be cached or stored on the client.

– Stateful
- Requests only contain the session ID, by which the server restores the corresponding session.

JSON Web Tokens (JWT)



\+ Security
- Apart from the initial exchange of credentials, they do not have to be sent again as long as the token remains valid.
- The token expiration time can be set (should usually be short-lived!).
- The token signature can be validated to ensure the integrity of the token (and thereby the data within the token).

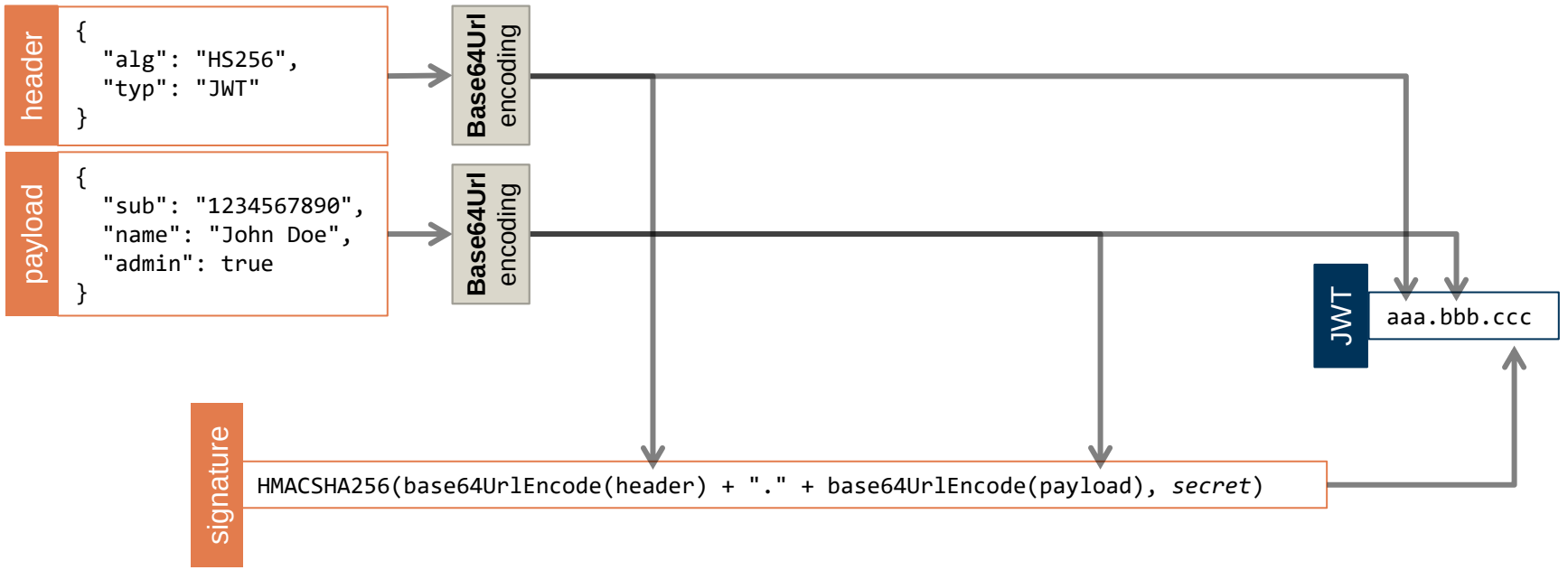
\+ Stateless
- The JWT contains everything required to determine and verify the user identity.

– Still vulnerable to attacks
- JWTs are often used as "bearer tokens", meaning the tokens can be used by anyone who possesses them.
  → Use TLS to securely transfer JWTs, as otherwise your application is vulnerable to man-in-the-middle and replay attacks.
- Contrary to sessions, JWTs cannot be easily revoked.
  → Use short expiration times or implement revocation functionality (could conflict with the goal of statelessness).
- If an outdated hashing algorithm is used, the private key used on the server side can be cracked and thereby the signature forged.
  → Use up-to-date hashing algorithm and keep your private key secure.

JSON Web Tokens (JWT)

Structure of a Token

- Emphasis on API authentication, especially for JSON-based REST APIs.
- JWTs are JSON objects consisting of 3 parts:
  - The **header** describes the type of the token as well as the hashing algorithm used for the signature.
  - The **payload** contains multiple claims (statements about an entity), e.g., username, expiration date, etc.
  - The **signature** is generated by the hash of the header, payload, and a secret held by the server. It ensures the integrity of the token.



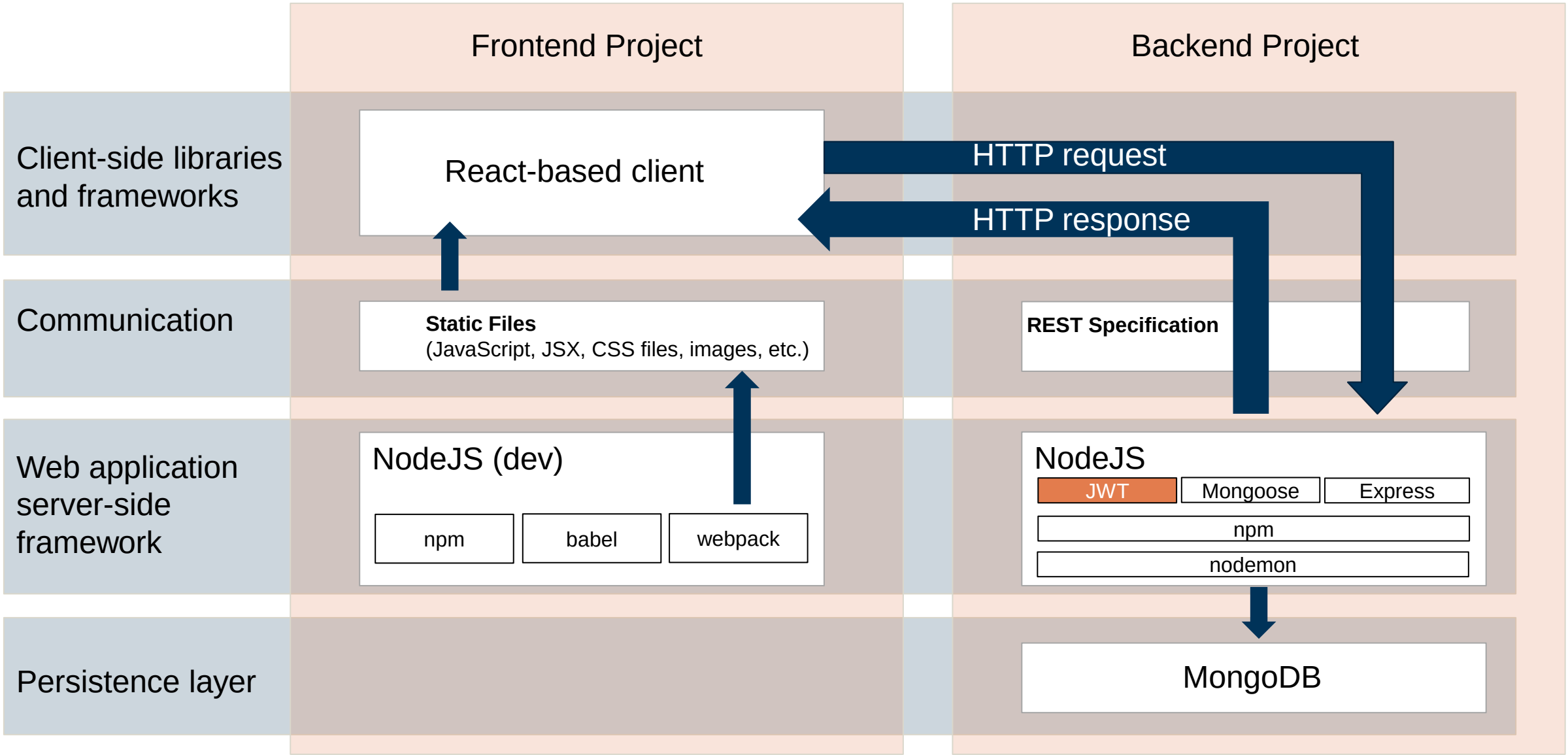
JWT

eyJhbGciOiJIUzI1NiI
sInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6Ik
pvaG4gRG9lIiwiYWRta
W4iOnRydWV9.
TJVA95OrM7E2cBab30R
MHrHDcEfxjoYZgeFONF
h7HgQ

https://jwt.io/

RFC 7519: https://datatracker.ietf.org/doc/html/rfc7519

Authentication With JWT



Authentication With JWT

Implementation as **middleware**

→ Middleware can be added to certain routes and executes independently before calling actual route processing function.

Examples of existing middleware:

- json **body-parser**: parses JSON-formatted request
- **cors**: Modifies headers to accept cross-server requests

Our middleware function parses the JWT token from the auth header.

If JWT is valid → injects user to the request payload → ready for e.g., further authorization.

Authentication With JWT

user.js example

```
"use strict";

const express        = require('express');
const router         = express.Router();

const middlewares    = require('../middlewares');
const AuthController = require('../controllers/auth');


router.post('/login', AuthController.login);
router.post('/register', AuthController.register);
router.get('/me', middlewares.checkAuthentication , AuthController.me);
router.get('/logout', middlewares.checkAuthentication, AuthController.logout);

module.exports = router;
```

middlewares.js: example of middleware for authentication

```
"use strict";

const jwt    = require('jsonwebtoken');

const config = require ('./config');
```

```
const checkAuthentication = (req, res, next) => {

    // check header or url parameters or post parameters for token
    const token = req.headers['x-access-token'];

    if (!token)
        return res.status(401).send({
            error: 'Unauthorized',
            message: 'No token provided in the request'
        });

    // verifies secret and checks exp
    jwt.verify(token, config.JwtSecret, (err, decoded) => {
        if (err) return res.status(401).send({
            error: 'Unauthorized',
            message: 'Failed to authenticate token.'
        });

        // if everything is good, save to request for use in other routes
        req.userId = decoded.id;
        next();
    });


};
```