Server-side JavaScript

C. Dallago, T. Goldberg, D. Nechaev, B. Rost, D. Schwartz, S. Wilzbach and G. Yachdav
Technische Universität München
Faculty of Informatics
Chair for Bioinformatics
What we will learn in this session:

- Node.js basics
- What you need to build your first back-end
- First hands-on experience with our example
Requests

http://example.dallago.us/

What’s at the root of example.dallago.us?

An index (index.html)
Requests II

http://example.dallago.us/script.js
http://example.dallago.us/image.png

The index needs script.js and image.png!
Requests III

<img src="http://example.dallago.us/image.png">
Requests IV

<img src="image.png">
Types of requests

- **GET**: Used to get/retrieve resources
- **POST**: Create and update resources (what web-forms do)
- **PUT**: Update and create resources (given an URL)
- **DELETE**: Delete resources
- … And more…
URLs

https://cell.map/assets/images/protein.png
URLs II

10.129.30.145

https://cell.map/  assets/  images/  protein.png
But what does JS have to do with all this?

Up until now:
- JavaScript for the front-end
- Interpret JavaScript in the browser’s console

Next:
- JavaScript for back-end development
- Interpret JavaScript as an ordinary computer program
- Use JavaScript to answer requests from the web
On a side note…
Question time:

1. How do you load a script in HTML?

2. What request does a browser perform when reading an <img> tag?

3. What is the difference between PUT and POST requests?
Question time:

1. How do you load a script in HTML?
   1. Using a `<script src="..">` tag
2. What request does a browser perform when reading an `<img>` tag?
   2. A GET request
3. What is the difference between PUT and POST requests?
   3. PUT is used mainly for update and needs specific URL
- JavaScript interpreter
- Single-core, ~1.8GB RAM
- Same language and concepts as for the front-end (event-based, asynchronous)
- Provides various libraries, for example to read/write files
- You can build:
  - Command-line scripts
  - APIs
  - Web applications with back- and front-end
Asynchronous, non-blocking execution

Traditional approach:

GET: File → getFile → open → read → send

GET: Data → getData → connect to DB → query → send

Node.js’s approach:

GET: File → Node V8 Engine → Event queue → Callback

GET: Data → Node V8 Engine → Event queue → Callback

Event queue:
- (getFile(file), open(file))
- (getData(db), connect(db))
- (open(file), read(file))
- (connect(db), query(db))
- (read(file), send(file))
- (send(file), none)
- (query(db), send(db))
- (send(db), none)
- Node Package Manager (NPM)
- Provides a large number of packages for:
  - Front-end
  - Back-end
  - Scripting
- Imagine node.js to be your smartphone and NPM to be your app store.
- As every app store: some apps are good, some are not (look at documentation!).
NPM growth

http://www.modulecounts.com
Node.js as a scripting language

ParsJS → [https://www.npmjs.com/package/parsjs](https://www.npmjs.com/package/parsjs)
Lets you transform a comma separated values file into an array of JSON objects:

```
name,surname,age
Christian,Dallago,23
Alex,Schmidt,25

[{
   "name": "Christian",
   "surname": "Dallago",
   "age": "23"
}, {
   "name": "Alex",
   "surname": "Schmidt",
   "age": "25"
}]
```
The JavaScript Object Notation (JSON)

- JSON is a data exchange format widely used in the web
- A JSON object looks like this:

```
{
   "name": "Christian",
   "surname": "Dallago",
   "age": "23"
}
```
The JavaScript Object Notation (JSON) II

In addition to Strings, JSON objects can contain all the primitive JS data types and compound type:

```json
{
  "type": "car",
  "price": 30440.32,
  "currency": "eur",
  "available": true,
  "locations": ["it", "es", "fr"],
  "reference": {
    "name": "Christian",
    "phone": "+3932984753"
  }
}
```
Defining dependencies to NPM package using package.json

In a node.js application, the NPM dependencies and the description of the application are in a file called package.json

```json
{
   "name": "example",
   "version": "1.0.0",
   "description": "Example",
   "main": "index.js",
   "author": "Christian Dallago <code@dallago.us>",
   "license": "ISC",
   "dependencies": {
      "express": "^4.10.1",
      "pug": "latest",
      "mongoose": "latest"
   }
}
```
Installing NPM packages using command line

$ npm install --global parsjs
Question time:

1. What can you use node for?

2. What can you store in a JSON object?

3. Where do you define dependencies in a node application?
Question time:

1. What can you use node for?
   1. Scripting, building backends and full web-apps
2. What can you store in a JSON object?
   2. All JavaScript primitive types and compound types
3. Where do you define dependencies in a node application?
   3. In the “dependencies” section of the package.json
Web-application components with node.js
Node.js as back-end - Request handling

• **Express.js**: de facto standard
• **koa**: simplified version of Express.js
• **Restify**: focus on REST APIs and based on Express.js
Node.js as back-end - Databases

- **SQLite, PostgreSQL** or **MySQL**: Relational databases
- **MongoDB, CouchDB**: JSON* document stores, a non-relational database type
- **Redis, LevelDB**: Key-value stores, another non-relational database type
Node.js as back-end - View Engines

Not needed if you build a REST API

- **Angular.js**: backed by Google, part of the Mongo+Express+Angular+Node (MEAN) stack
- **React**: developed by Facebook
- **Pug** (previously Jade): powerful and simple templating engine
- **Ember**: another choice among small communities of developers
Question time:

1. Why would you use a non-relational database to prototype?

2. In what order do you need components to build your application?
Question time:

1. Why would you use a non-relational database to prototype?
   1. No schema, no relations, simple to conceptualize

2. In what order do you need components to build your application?
   1. Request handler
   2. Database
   3. View Engine
An (actually; two) example application(s)
Node.js as back-end in our example
Node.js as back-end in our example II

Views are going to be explained in the next session. Let’s focus on building an API.
Reference examples

First code example (API): http://example.dallago.us/public/api.zip
Complete code example: http://example.dallago.us/public/complete.zip

Running example: http://example.dallago.us

The example uses the well known **Model View Controllers (MVC)** pattern:

- Models (the data objects),
- Views (the way this data is presented) and
- Controllers (the logic that combines data and calculations to produce meaningful Views).
First example folder structure

```
/ app
  controllers
    relationships.js
  models
    relationships.js
  views
config.json
database.js
index.js
package.json
router.js
```

Folders
Server-side JavaScript
Complete example folder structure

- Folders
- Server-side JavaScript
- Views
- Front-end JavaScript
- Front-end Style (CSS)
How to get the examples running

1. Download the zip files
2. Unzip the files
3. Make sure you have node.js and NPM installed (https://nodejs.org)
4. Use the command-line to navigate to the folder where index.js is
5. Run the command npm install (this will install the dependencies and create a new folder in the current directory called “node_modules”)
6. Run the command node index.js
7. Navigate from your browser to http://localhost:3000/api/relationships
Controllers and Models
- Framework that allows to react on incoming requests and send some response
- Provide extendable middle wares and layers
- Can be extended through other packages
Express.js in our first example application

/ app
  controllers
    relationships.js
  models
    relationships.js
  views
    config.json
    database.js
    index.js
    package.json
    router.js
const express = require('express');
const app = express();

// Set 'port' value to either an environment value PORT or 3000
app.set('port', process.env.PORT || 3000);

// Create router
var router = require('./router');

// Router listens on / (root)
app.use('/', router);

// Start listening for requests on the port defined earlier
app.listen(app.get('port'), function(){
    console.log("Express server listening on port " + app.get('port'));
});
Express.js: router.js + controllers/relationships.js

```javascript
// API
router.get('/api/relationships', relationshipsController.relationships);

relationships: function(request, response) {
    relationshipsModel.find(function(error, results) {
        if (error) {
            response.status(500).send(error);
        } else {
            response.status(200).send(results);
        }
    });
```
- Document-oriented database management system
- Non-relational system
- Stores BSON, that is: Binary-JSON
- Can store images, music, as well as ordinary JSON objects
- Out of the box it doesn’t enforce a schema
MongoDB

JSON:
```
{
  "name": "Christian",
  "surname": "Dallago",
  "age": "23"
}
```

MongoDB entry:
```
{
  "_id": {
    "$oid": "57657fe0dcba0f1e46d2aa4b"
  },
  "name": "Christian",
  "surname": "Dallago",
  "age": "23"
}
```
- To connect to a mongo database you need a “driver”
- Allows you to:
  - Define Models (enforce some type of schema)
  - Perform queries (find, create, update, remove items)
  - Connect to databases and handle connection events
Mongoose in our first example application

/ app
  controllers
    relationships.js
  models
    relationships.js
  views
    config.json
    database.js
    index.js
    package.json
    router.js
const mongoose = require('mongoose');

const relationshipsModel = mongoose.model('relationships', {
  source: {
    type: String,
    require: true
  },
  target: {
    type: String,
    require: true
  },
  weight: {
    type: Number,
    require: true
  }
});

module.exports = relationshipsModel;
const relationshipsModel = require('../models/relationships'));

module.exports = {
    relationships: function(request, response){
        relationshipsModel.find(function(error, results){
            if(error){
                response.status(500).send(error);
            } else {
                response.status(200).send(results);
            }
        });
    }
};
Mongoose: What else can you do?

You can store new items in the database

```javascript
relationshipsModel.create({
    source: "me",
    target: "you",
    weight: 6
}, function(error, newItem){
    console.log("Created:", newItem);
});
```
Mongoose: What else can you do?

You can remove items from the database

```javascript
relationshipsModel.findOneAndRemove({
    source: "me"
}, function(error, removedItem){
    console.log("Removed:", removedItem);
});
```
Mongoose: What else can you do?

You can update items in the database

```javascript
relationshipsModel.update({
  source: "me"
}, {
  $set: {
    target: "Tommen"
  }
});
```

Will update only the first document matching the query
Mongoose: CRUD

What we have just seen has been the CRUD stack:

- **Create** → `Model.create()`
- **Read** → `Model.find()`
- **Update** → `Model.update()`
- **Delete** → `Model.remove()`
Question time:

1. Why do you pass the prototype of a function to a request handler?

2. Do you have to implement the CRUD stack in your application?
Question time:

1. Why do you pass the prototype of a function to a request handler?
   1. Because we want the function to be executed every time a request comes, and not service the output of one function call (on startup) for every request.

3. Do you have to implement the CRUD stack in your application?
   2. Depends!!
Views
- Ia view engine
- Gives us a way to define reusable and extendable HTML components
- Imagine your web page as a set of pieces: navigation, menus, content, footer,…
- Uses a simplified syntax
- Can use control statements (like each and if) and variables passed to the view
Pug:
Where do you find it?

/ app
  controllers
    relationships.js
    frontend.js
  models
    relationships.js
  views
    base.pug
    home.pug
    navigation.pug
    visualization.pug

public
  libs
    d3.js
  visualization
    script.js
    style.css
  global.css
  config.json
  database.js
  index.js
  package.json
  router.js
Pug: home.pug + base.pug + navigation.pug

```
html
  head
    title= title
    link(rel='stylesheet', href='/public/style.css')
    meta(charset="utf-8")
  body
    .wrapper
      include navigation
      block content

  extends base
  block content
    h1 Here is a title
    p A nice Paragraph
```

ul
  li
    a(href="/" title="Home") Home
Pug: home.pug + base.pug + navigation.pug

<html>
  <head>
    <title>Home</title>
    <link rel="stylesheet" href="/public/style.css"/>
    <meta charset="utf-8"/>
  </head>
  <body>
    <div class="wrapper">
      <ul id="navigation">
        <li><a href="/">Home</a></li>
      </ul>
      <h1>Here is a title</h1>
      <p>A nice Paragraph</p>
    </div>
  </body>
</html>
Question time:

1. What are views used for?

2. What is the difference between *extending* and *importing*?

3. Give me two examples of applications in biology where you need views and two where you don’t need them
Question time:

1. What are viewes used for?
   2. Extend: You extend part of existing code at a specific location.
   3. Import: You take all the code in the file and paste it in that location.

3. What is the difference between extending and importing?
Question time
Demo time