

# Full Stack JS Server Side

**R. Mark Volkmann**  
Object Computing, Inc.  
<http://objectcomputing.com>  
Email: [mark@objectcomputing.com](mailto:mark@objectcomputing.com)  
Twitter: @mark\_volkman  
GitHub: mvolkmann



OCI | TRAINING

# Outline

- **REST** overview
- **Express** for serving files and implementing REST services
- **cURL** and **Postman** for testing REST services
- **Relational databases** overview
- **MySQL** overview
- **Demo app**
- **Hands on exercises**

## Preparation Steps

```
git clone https://github.com/mvolkmann/react-tour-of-heroes
cd react-tour-of-heroes
cd server
npm install
cd ../client
npm install
```



# Web Application Parts

- Web applications primarily have two parts

- Front-end / client-side

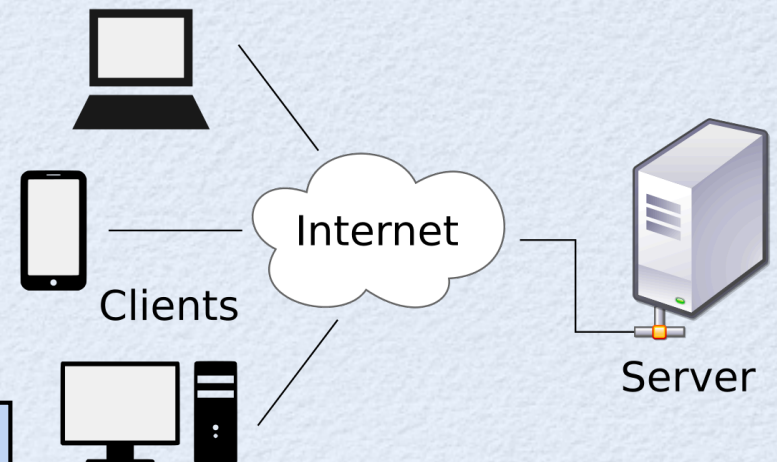
- dynamic generation of HTML
- REST calls to retrieve and modify persisted data
- state management
- page routing

There are many options.  
Most use JavaScript.  
Our example app uses React.

- Back-end / server-side

- serve static files
- implement REST services
- retrieve and modify persisted data

There are many options that use many programming languages.  
This workshop focuses on writing in JavaScript using Node.js and Express.





# REST Overview

- Stands for **RE**presentational **S**tate **T**ransfer
- An architectural style, not a standard or API
- Described in Roy Fielding's dissertation in 2000 (chapter 5)
  - <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- Main ideas
  - a software component requests a "**resource**" from a service
    - by supplying a resource identifier and a desired media type
    - resource identifier can be a URL and request can be made using Ajax
  - a "**representation**" of the resource is returned
    - a sequence of bytes and metadata to describe it
    - can be JSON, name/value pairs in HTTP headers, an image, ...
    - can contain identifiers of other resources
  - obtaining this representation causes the software component to "**transfer**" to a new "**state**"



# Typical REST

- Relies on HTTP(S)
- Uses carefully selected URLs to identify resources
- Uses HTTP methods for specific operations a.k.a. HTTP verbs

primarily for Create-Retrieve-Update-Delete (**CRUD**) operations

  - **GET** - retrieves representation of existing resource identified by URL
  - **POST** - creates new resource using data in body
    - URL of new resource is not yet known
    - often the request URL identifies an existing parent resource
    - return URL of new resource in **Location header** and **status 201 Created**

ex. POST to URL for a music artist, supplying data describing a new CD; URL of new CD resource is returned
  - **PUT** - updates existing resource identified by URL using data in body
    - URL of resource is already known
    - like POST, body contains entire description of resource
    - can also create a new resource if its URL is already known
  - **PATCH** - like PUT, but body only describes changes
    - other properties retain their current value
  - **DELETE** - deletes existing resource identified by URL
  - **POST** - for everything else (non-CRUD operations)



# Express

<https://expressjs.com/>

- “A minimal and flexible Node.js web application framework”
- Provides HTTP request routing and redirection, view rendering, and more
  - routing associates an HTTP method and URL pattern with a function to handle it
  - view rendering typically refers to server-side HTML generation
- Extends request and response objects from Node `http` module
  - with extra properties and methods
- To install
  - `npm install express`
- API documentation
  - <https://expressjs.com/en/4x/api.html>
  - click these in upper-right →
- Example code
  - <https://github.com/expressjs/express/tree/master/examples>

`express()`  
Application  
Request  
Response  
Router



# Express Use Cases

- Serve static files
  - HTML, CSS, JavaScript, images, audio, video, PDFs, ...
- Serve dynamically generated content
  - including HTML
- Act as a REST server
  - most popular use case and our focus

# Express Does Not Provide

- Help with defining and using model objects
- Database access
- Object-Relational Mapping (ORM)
- HTML templating
- User authentication
- Security

Many other Node modules  
provide these features!



# Middleware ...

- Not a term unique to Express, but it has a specific definition

- A function that takes

- a request object
- a response object
- a **next** function

```
function myMiddleware(req, res, next) {  
  // Maybe do something with req.  
  // Maybe do something with res.  
  // Either call next or  
  // cause response to be sent to client.  
}
```

If it doesn't call **next** then technically it isn't "middle"-ware.

- A single request can be processed by a sequence of these

- "middleware stack"
- run in order registered

- Calling **next** allows next middleware in stack to run

- supports performing asynchronous actions where the next middleware cannot start until the current one completes an asynchronous action
- to report an error and skip remaining middleware, pass an **Error** object to **next**, **do not throw**

- ex. `next(new Error('something bad happened'));`



# ... Middleware

- Examples include logging, authentication, authorization, cookie parsing (**cookie-parser**), request body parsing (**body-parser**), and response compression
- The only included middleware is **express.static**
  - uses npm package **serve-static** to serve files from a given directory
- Many third party middleware modules are available
  - good source is <http://expressjs.com/resources/middleware.html>



# Serving Static Files

```
const express = require('express');  
  
const app = express();  
  
// Can have multiple lines like the following to specify  
// each directory to be searched for static files.  
app.use(express.static('public'));  
  
const PORT = 1234;  
app.listen(PORT, () => console.log('ready'));
```

to see the file  
public/foo/bar.html  
browse  
<http://localhost:1234/foo/bar.html>



# Simple Route Handling

```
const express = require('express');  
  
const app = express();  
  
app.get('/',  
  (req, res) => res.send('Hello!'));  
  
const PORT = 1234;  
app.listen(PORT, () =>  
  console.log('browse http://localhost:' + PORT));
```

a middleware function

# Using Middleware

- To register a middleware

`app.use(myMiddleware);` or `app.use(path, myMiddleware);` to only run for matching paths

- can run before or after handlers for specific paths
- Each is called in the order in which it was registered

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

const app = express();

app.use(morgan('short')); // request logging

app.use(express.static('public'));

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

const port = 3000;
app.listen(port, () => {
  console.log('listening on port', port);
});
```

← a middleware function

browse <http://localhost:3000/hello>



# Routes

- To listen for requests using a specific HTTP method, `app.method(path, middlewareFn)`
  - `path` is a string or regular expression
  - `method` is one of `get`, `patch`, `post`, `put`, `delete`, or one of the many less commonly used HTTP methods
- Can use to implement a REST API
- Can register more than one middleware for same path
  - run in order registered

referred to as  
defining "routes"

# Request Data Passing Options

- Query parameters

- at end of URL after ?
- ex. `http://cars.com/car?make=MINI&model=Cooper&year=2015`

- Path parameters

- positional, slash-separated data at end of URL
- ex. `http://cars.com/car/MINI/Cooper/2015`

- Body

- can be text or binary data
- often JSON text
- specify format in **Content-Type** request header (ex. `application/json`)

- ex. 

```
{  
  "make": "MINI",  
  "model": "Cooper",  
  "year": 2015  
}
```



# Query Parameters

- At end of URL after ?
  - ex. `http://cars.com/car?make=MINI&model=Cooper&year=2015`
- Gathered in an object stored in `query` property of request object that is passed to middleware
- Can use destructuring to get values

```
const {make, model, year} = req.query;
```
- When query parameters are repeated, the value is an array of strings instead of a string
  - ex. `http://market.com/order?fruit=apple&fruit=banana&dairy=milk`
  - `req.query.fruit` is `['apple', 'banana']`
  - `req.query.dairy` is `'milk'`



# Path Parameters

- Positional, slash-separated data at end of URL
  - ex. `http://cars.com/car/MINI/Cooper/2015`
- Include names preceded by colons in route path
  - ex. `"/car/:make/:model/:year"`
- Path parameters are gathered in an object stored in `params` property of request object that is passed to middleware
- Can use destructuring to get values

```
const {make, model, year} = req.params;
```



# Request Bodies

- To parse JSON
  - `app.use(express.json());`
  - used when request **Content-Type** header is `'application/json'`

- To parse plain text
  - `const bodyParser = require('body-parser');`
  - `app.use(bodyParser.text());`
  - used when request **Content-Type** header is `'text/plain'`

can pass an options object to the `json`, `text`, and `urlencoded` methods; see <https://www.npmjs.com/package/body-parser>

- To parse HTML form data
  - `app.use(express.urlencoded({extended: true}));`
  - used when request **Content-Type** header is `'application/x-www-form-urlencoded'`

when `extended` is `true`, the `qs` library is used which "allows for rich objects and arrays"  
when `extended` is `false`, the `querystring` library is used



# Request Properties

- **body**
  - `body-parser` middleware populates this when it parses a request body
- **ip**
  - IP address of sender
  - could use as part of authentication solution
- **params**
  - object that holds properties extracted from URL path
  - ex. `/car/:make/:model/:year`
- **query**
  - object that holds parsed query parameters
  - ex. `http://cars.com/car?make=MINI&model=Cooper&year=2015`
- and many more



# HTTP Status Codes

- Levels
  - **1xx** - informational
  - **2xx** - **success**
  - **3xx** - redirection
  - **4xx** - **client errors**
  - **5xx** - **server errors**
  - **6xx** - unofficial
- Commonly used
  - **200** - OK
  - **304** - Not Modified
  - **400** - Bad Request
  - **404** - Not Found
  - **500** - Internal Server Error
- Less-commonly used
  - **201** - Created; successfully created a resource
  - **202** - Accepted; request will be successful; but will be handled later asynchronously
  - **204** - No Content; success with no response (ex. delete)
  - **301** - Moved Permanently; resource URL changed; new URL is in `Location` header
  - **401** - Unauthorized; can't perform action because user hasn't authenticated (logged in)
  - **403** - Forbidden; user has authenticated, but doesn't have permission for the action
  - and many more

set in response by calling  
`res.status(code)`  
(see next slide)



# Response Methods

- To send data

- `res.send(data);`
  - where `data` is a string, object, array, or `Buffer`
- `res.json(data);`
  - converts `data` to JSON by calling `JSON.stringify(data)`

Content-Type response header is automatically set to `text/html` for strings and `application/json` for objects

- To send file content

- `res.sendFile(filePath);`

- To set response headers

- `res.set(name, value)` or `res.set(obj)` to set many in one call

- To set response status

- `res.status(code)`
- defaults to 200 (success), 404 if no route matches the URL, and 500 if any middleware throws an error
- can chain a call to `send`, `json`, or `sendFile` after this

example  
`res.status(400).send('bad car make');`

- and many more



# Automatic Server Restart

- During iterative server development it's convenient for the server to restart automatically when source files change

- Two good solutions for Node

- **nodemon**

- <http://nodemon.io/>
- good option when not transpiling code
- `npm install -g nodemon`
- use `nodemon` command in place of `node` command
- if server crashes, waits for file changes before restarting

```
"start-dev": "nodemon src/index.js",
```

- **babel-watch**

- <https://www.npmjs.com/package/babel-watch>
- good option when using Babel to transpile code

```
"start-dev": "babel-watch --message 'restarting' --ignore node_modules -- src/index.js",
```

# Routers

- A mechanism to organize a server into related routes with each group defined in a separate source file
- Example

```
import express from 'express'; index.js
import fooRouter from './foo-router';

const app = express();

app.use('/foo', fooRouter);
// Can register more routers.

app.listen(3001);
```

```
import express from 'express';
const router = express.Router();

// Register middleware that is only used
// for routes defined on this router.
router.use(someMiddleware);

function barHandler(req, res) {
  res.send('baz');
}
router.get('/bar', barHandler);

export default router;
```

url to access this is  
<http://localhost:3001/foo/bar>



# Using HTTPS ...



- Need an SSL certificate
  - many options
- Option #1
  - generate private key - `openssl genrsa -out privatekey.pem 1024`
  - generate signing certificate request - `openssl req -new -key privatekey.pem -out request.pem`
    - answer questions
  - request a certificate using <https://letsencrypt.org>
- Option #2
  - use openssl

```
openssl req -x509 \  
-sha256 \  
-nodes \  
-newkey rsa:2048 \  
-days 365 \  
-keyout localhost.key \  
-out localhost.crt
```

based on an answer at  
[http://stackoverflow.com/questions/8169999/  
how-can-i-create-a-self-signed-cert-for-localhost](http://stackoverflow.com/questions/8169999/how-can-i-create-a-self-signed-cert-for-localhost)

# ... Using HTTPS



- Can configure Express to support both HTTP and HTTPS

```
const express = require('express');
const fs = require('fs');
const http = require('http');
const https = require('https');

const app = express();

// Register middlewares and specify routes.

const HTTP_PORT = 80;
const HTTPS_PORT = 443;

const httpsOptions = {
  key: fs.readFileSync('.localhost.key'),
  cert: fs.readFileSync('.localhost.crt')
};

http.createServer(app).listen(HTTP_PORT);
https.createServer(httpsOptions, app).listen(HTTPS_PORT);
```

browsing `https://localhost`  
will use port 443 by default

start server with  
`sudo node server.js`



# CORS

- Stands for “Cross-Origin Resource Sharing”
- Allows clients to use services from domains other than the one from which they were served
  - host and/or port differs
- Must enable in servers
- Easiest way for Node servers to support this
  - `npm install cors`
    - see <https://www.npmjs.com/package/cors>
  - can configure to only allow requests from specified domains
    - allows all by default
  - can configure to only allow specified HTTP methods
    - allows all by default

```
import cors from 'cors';  
app.use(cors());
```



# cURL



- “Client for URLs” to use with Windows, install Cygwin (<http://cygwin.com>) or see <http://curl.haxx.se>
- Command-line tool for evaluating many kinds of internet protocol requests
  - HTTP, HTTPS, FTP, LDAP and many more
  - `curl [options] [urls]`
- URLs
  - can guess protocol; not necessary to include `http://` prefix
  - encode special characters (URI encoding, a.k.a. percent-encoding)
    - ex. %20 for spaces, %2F for slashes search for “percent-encoding” in Wikipedia
  - ampersands separating query parameters must be preceded by backslashes
    - ex. `http://foo.com:1234/bar/baz?kind=lucky\&value=7`
- Exit codes
  - there are a large number of them (~ 80) that can be tested for in scripts
  - ex. 3 = URL malformed; 6 = couldn't resolve host; 7 = failed to connect to host
  - see man page for list (`man curl`)



# cURL Options ...



- **--help** or **-h** - output help
- **--request** or **-X *verb*** - HTTP verb to be used
  - ex. **GET** (default), **POST**, **PUT**, **PATCH**, **DELETE**
- **--header** or **-H '*name: value*'** - request header
  - ex. **-H 'Content-Type: application/json'**
  - use once for each request header
- **--data** or **-d '*data*'** - body data
  - use with HTTP **POST**, **PUT**, and **PATCH**
  - changes default verb to **POST**
  - to get data from a text file, **-d @*file-path***
  - to get data from a binary file, **--data-binary @*file-path***
  - to get data from stdio, **-d @-**; after entering data, press return and ctrl-d

space between option  
and value is optional

option values containing  
special characters must  
be surrounded with  
single or double quotes



# ... cURL Options



- **--user** Or **-u** *username[:password]* - for authentication
  - will prompt for password if omitted
- **--include** or **-i** - include response headers in output
- **--head** or **-I** - send a **HEAD** request
  - same as **GET**, but only returns response headers, not body
  - **-XHEAD** doesn't work
- **--location** or **-L** - for redirection
  - if a 3xx status is returned indicating redirection (ex. data has moved), this reissues the command with the new location
- **--write-out** Or **-w** '*format*' - for additional output after successful response
  - ex. to add a newline, **-w** '**\n**'
  - many variables can be included in format using **%{name}**; see man page for list
  - ex. **-w** '**\n%total time = {time-total} seconds\n**'
- **--output** Or **-o** *file-path* - write output to a file instead of stdout



# curl Examples for Demo App

- Get uptime
  - `curl localhost:3001`
- Get all heroes
  - `curl localhost:3001/hero`
- Get hero by id
  - `curl localhost:3001/hero/id`
- Create hero
  - `curl -X POST -H "Content-Type: application/json" \`  
`-d '{"name": "name"}' localhost:3001/hero`
- Update hero
  - `curl -X PUT -H "Content-Type: application/json" \`  
`-d '{"name": "new-name"}' localhost:3001/hero/id`
- Delete hero by id
  - `curl -X DELETE localhost:3001/hero/id`



# Postman ...

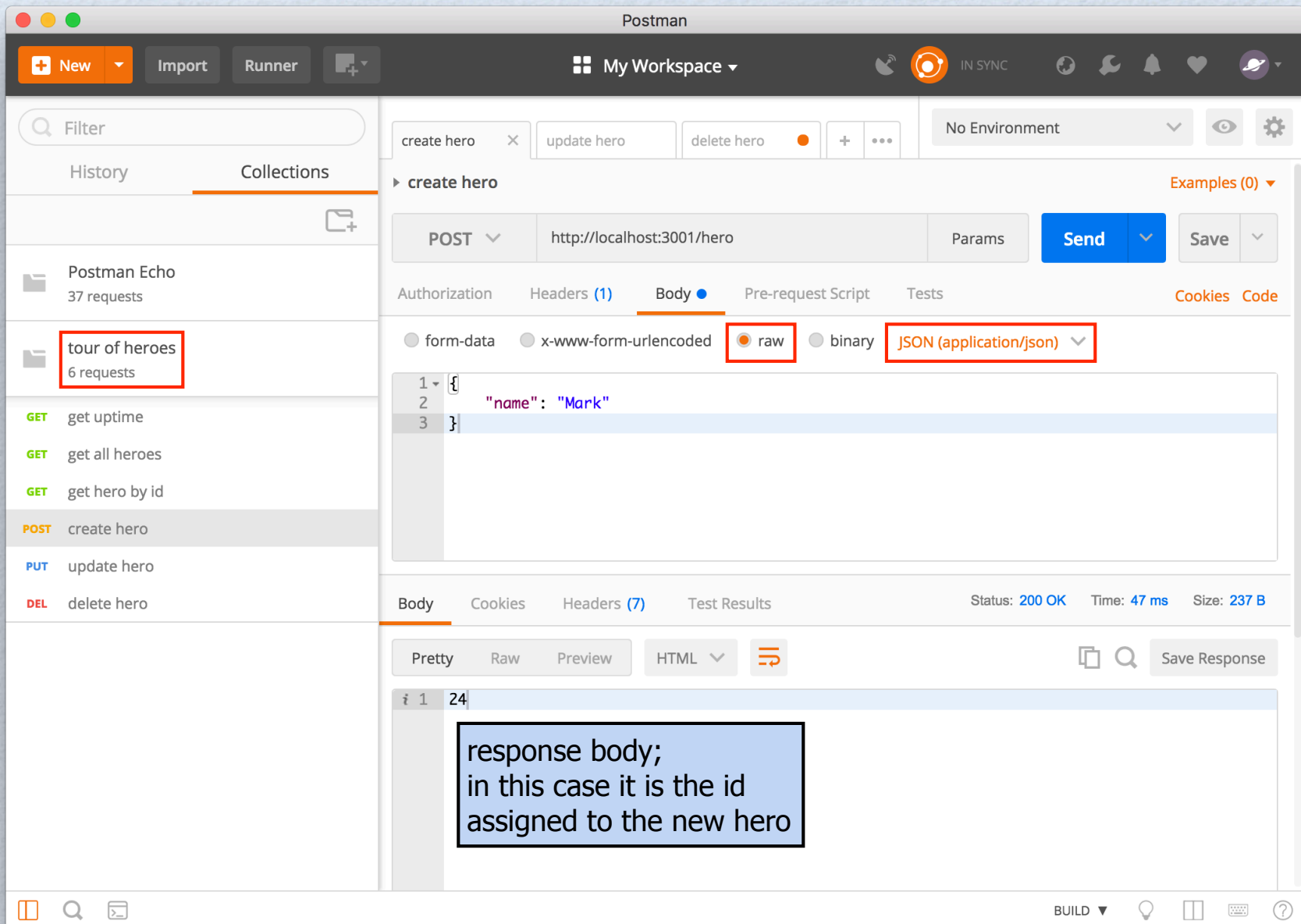
<https://www.getpostman.com/>

- Application for testing REST APIs
- Free for basic use cases
- Enter and send HTTP requests
- Organize requests in “collections”
- Name and describe each saved request
- Select a previous request and resend
- To include a JSON request body
  - add **Content-Type** header with a value of **application/json**
  - add body with type “raw” and “JSON” (see next slide)





# ... Postman



# Try It!

- Implement a simple REST service
  - add numbers passed in request body JSON array

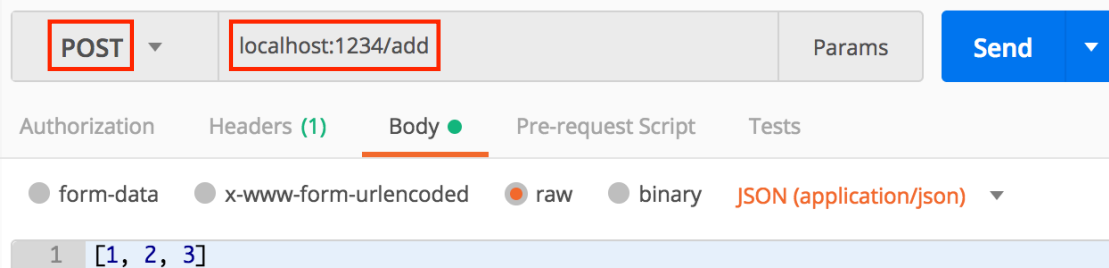
- Steps

- create directory **express-demo** and cd to it
- **npm init**
  - accept all defaults; creates **package.json**
- **npm install express body-parser**
- **npm install -g nodemon**
- edit **package.json** scripts

```
"scripts": {  
  "start": "nodemon index.js"  
},
```

- create **index.js**
- **npm start**
- test using Postman

```
const express = require('express');  
const bodyParser = require('body-parser');  
  
const app = express();  
  
app.use(bodyParser.json());  
  
app.post('/add', (req, res) => {  
  const numbers = req.body;  
  const sum = numbers.reduce(  
    (sum, n) => sum + n;  
  );  
  res.send(String(sum));  
});  
  
const PORT = 1234;  
app.listen(PORT, () => console.log(  
  'browse http://localhost:' + PORT));
```





# Relational Databases

- There are many ways to persist data on a server
- Relational databases are the most popular option and our focus
- Many npm packages are available for specific databases
- Some npm packages support multiple databases
  - **Sequelize** supports Microsoft SQL Server, MySQL, PostgreSQL, and SQLite
    - <http://docs.sequelizejs.com/>
- We will focus on MySQL and use **mysql-easier**
  - <https://www.npmjs.com/package/mysql-easier>

# SQL Review



- Language for interacting with relational databases
  - details sometimes vary between databases

## Create table

```
create table table-name (  
    column-name type modifiers,  
    ...  
)
```

examples of modifiers include  
not null and primary key

## Delete table

```
drop table table-name
```

## Insert rows

```
insert into table-name  
    (column-name, ...)  
values (value, ...)
```

## Update rows

```
update table-name  
set column-name=value, ...  
where condition
```

## Delete rows

```
delete from table-name  
where condition
```

## Retrieve rows

```
select [unique] column-name, ...  
from table-name, ...  
where condition  
order by column-name, ...
```



# SQL Joins



**person** table

id	name	role	address_id
1	Mark Volkmann	developer	19
2	Matt Bremehr	sales	20

**address** table

id	street	city	state	zip
19	644 Glen Summit	St. Charles	MO	63304
20	123 Some Street	Some City	MO	12345

- Example "inner join" query  

```
select p.name, a.zip
from person p, address a
where p.address_id = a.id
```

- JSON result is  
[  
 {"name": "Mark Volkmann", "zip": 63304},  
 {"name": "Matt Bremehr", "zip": 12345}  
]



# MySQL

<https://www.mysql.com/>

- To verify version installed

- `mysql --version`

npm scripts shown here must be run from `react-tour-of-heroes/server` directory

- To start server

- `npm run dbstart`

- To stop server

- `npm run dbstop`

Don't use these in Windows since the MySQL server is installed to run as a Windows service.

- To populate `tour_of_heroes` database with `hero` table and initial data

- `npm run dbsetup`

- To enter interactive mode where SQL statements can be entered

- `npm run dbi`

can run commands like `show databases`, `show tables`, and `describe table-name`





# mysql-easier ...

<https://www.npmjs.com/package/mysql-easier>

- Makes it very easy to interact with MySQL databases
- `npm install mysql-easier`
- Example

For **PostgreSQL**, see  
<https://www.npmjs.com/package/postgresql-easy>

```
const mse = require('mysql-easier');
const config = require('./config.json');
mse.configure(config);

async function demo() {
  const conn = await mse.getConnection();
  const newId = await conn.insert(
    'food', {name: 'Apple', color: 'Red'});
  const item = await conn.getById('food', newId);
  console.log(item.name) // Apple
  conn.deleteById('food', newId);

  // If finished with connection ...
  await conn.done();

  // If ready to exit app ...
  await mse.endPool();
}

demo();
```

config.json

```
{
  "database": "myDbName",
  "debug": false,
  "host": "localhost",
  "password": "root",
  "port": 3306,
  "user": "root"
}
```

# ... mysql-easier

- Emphasizes tables with columns named "id" that are auto-incrementing primary keys
- **MySQLEasier** methods include
  - **configure**
  - **createPool, endPool** ← for pools of connections that support concurrent operations
  - **createConnection** (non-pooled), **getConnection** (from global pool)
- **MySQLConnection** methods include
  - **deleteAll, deleteById**
  - **getAll, getById**
  - **insert**
  - **query** - accepts any SQL statement and parameters
  - **updateById, upsert** **upsert** updates if present and inserts otherwise
  - **transaction**
  - **done** - closes connection; if from a pool, returns it to pool



# Wrap Up

- Server-side recommendations
  - Node.js
  - Express
  - Babel
  - MySQL and mysql-easier
  - PostgreSQL and postgresql-easy
- Recommendations for server and client
  - ESLint
  - Prettier

# Demo App

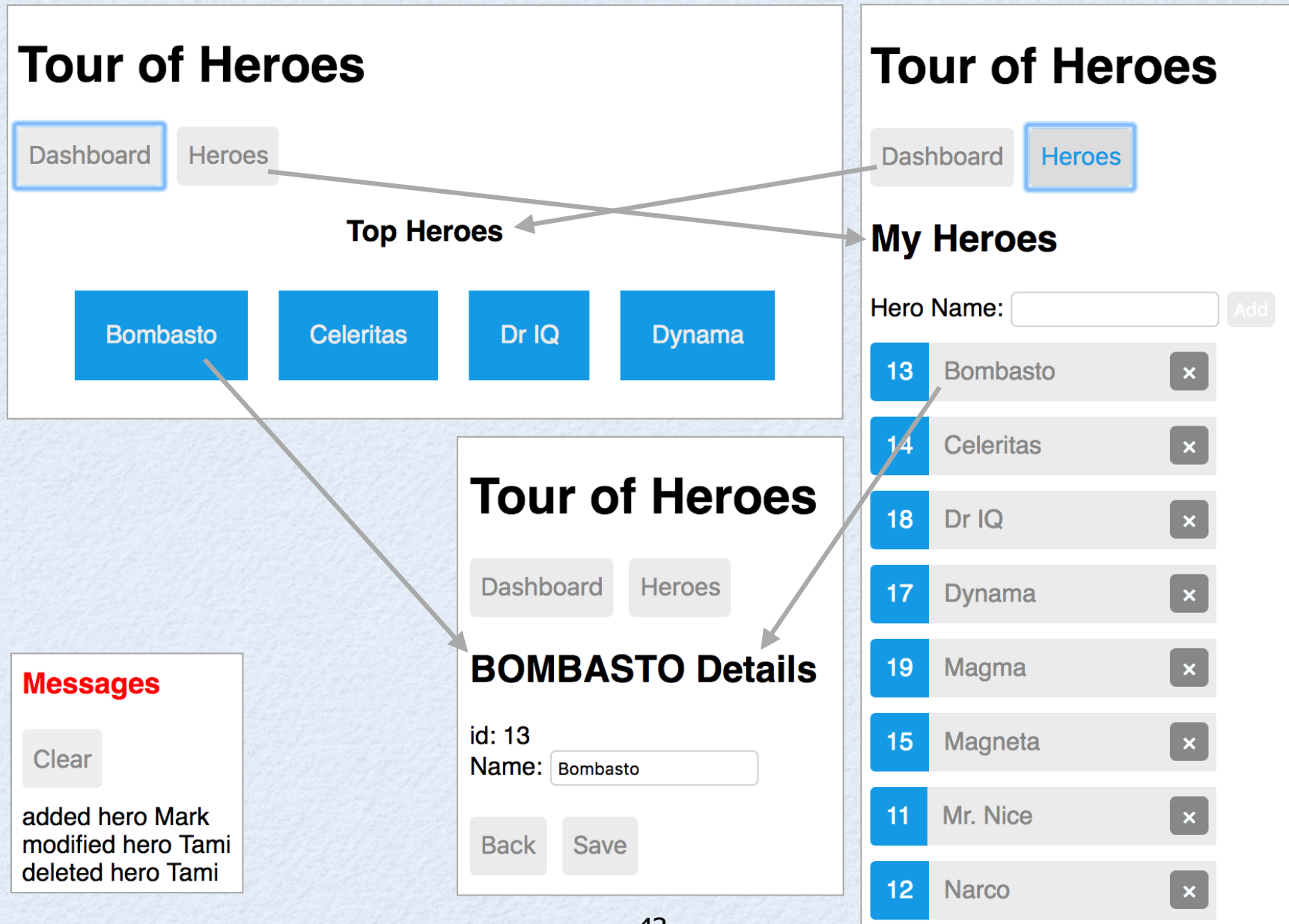
- “Tour of Heroes” is the app used in the Angular tutorial
  - <https://angular.io/tutorial>
- Let’s walk through how it can be implemented using Node.js, Express, MySQL, and React
  - code is at <https://github.com/mvolkmann/react-tour-of-heroes>
- Contains **client** and **server** directories
- Uses Flow for type checking



# Steps to Run Demo App

- `git clone https://github.com/mvolkmann/react-tour-of-heroes.git`
- In terminal window #1
  - `cd` to repo directory
  - `cd server`
  - `npm install`
  - `npm run dbstart` (if not in Windows)
  - `npm run dbsetup`
  - `npm run start-dev`
- In terminal window #2
  - `cd` to repo directory
  - `cd client`
  - `npm install`
  - `npm start`
- In any browser (except IE), browse localhost:3000

# Demo Screenshots





```

react-tour-of-heroes/
| .git/
| client/
|   | node_modules/
|   | public/
|   |   | favicon.ico
|   |   | index.html ★
|   |   | manifest.json
|   | src/
|   |   | dashboard/
|   |   |   | dashboard.css
|   |   |   | dashboard.js
|   |   |   | dashboard.scss
|   |   | hero-detail/
|   |   |   | hero-detail.css
|   |   |   | hero-detail.js
|   |   |   | hero-detail.scss
|   |   | hero-list/
|   |   |   | hero-list.css
|   |   |   | hero-list.js
|   |   |   | hero-list.scss
|   |   | messages/
|   |   |   | messages.css
|   |   |   | messages.js
|   |   |   | messages.scss
|   |   | util/
|   |   |   | error-util.js
|   |   |   | rest-util.js
|   |   | App.css
|   |   | App.js ★
|   |   | App.scss
|   |   | index.js ★
|   |   | initial-state.js
|   |   | reducers.js
|   |   | share.css
|   |   | share.scss
|   |   | types.js
|   |   | .eslintrc.json
|   |   | .flowconfig
|   |   | package-lock.json
|   |   | package.json

```

# Client-side Files

We will cover this in much more detail in the "Client" section of the workshop!

- `.css` files are generated from `.scss` files using Sass CSS preprocessor
- React components are **App, Dashboard, HeroDetail, HeroList, and Messages**
- Uses `redux-easy` npm package to simplify use of Redux for state management
  - <https://www.npmjs.com/package/redux-easy>
- Uses Fetch API to make REST calls
  - encapsulated in `src/util/rest-util.js`
- **public/index.html** - provides element where **App** component will render
- **src/index.js** - configures use of Redux using `redux-easy`
- **src/App.js** - topmost component; decides what to render based on `route` state property



# Server-side Files

```
react-tour-of-heroes/  
| .git/  
| client/  
| screenshots/  
| server/  
| | flow-typed/  
| | node_modules/  
| | src/  
| | | util/  
| | | crud-router.js  
| | | crud-router.test.js  
| | | database.js ★  
| | | hero-router.js ★  
| | | hero-router.test.js  
| | | index.js ★  
| | .babelrc  
| | .eslintrc.json  
| | .flowconfig  
| | config.json  
| | ddl.sql  
| | package-lock.json  
| | package.json  
| .gitignore  
| .prettierrc  
| README.md
```

- **src/index.js** - configures use of many npm packages
  - **body-parser** for parsing request bodies containing JSON and text
  - **cors** for cross-origin resource sharing
  - **express** for middleware and REST service routes
  - **express-healthcheck** for uptime stats
  - **morgan** for request logging
- **src/database.js** - configures use of MySQL using **mysql-easier**
- **src/hero-router.js**
  - used by **src/index.js**
  - configures Express routes related to heroes
  - implements REST services
  - see functions **deleteHero**, **getAllHeroes**, **getHeroById** (not used by UI), **postHero** (to create new), and **putHero** (to modify existing)
  - see **wrap** function that gets database connection and provides error handling



# REST Calls

to `http://localhost:3001`

- **client/src/util/rest-util.js**
  - exports functions that use the Fetch API to make REST calls
  - includes `deleteResource`, `getJson`, `getText`, `patchJson`, `post` (no body), `postJson`, and `putJson`
- **GET /** - health check that returns uptime
- **POST /hero** - creates a new hero **Create**
  - see `postJson` call in `client/src/hero-list/hero-list.js`
- **GET /hero** - gets JSON array of all heroes in database **Retrieve**
  - see `getJson` call in `client/src/index.js`
- **PUT /hero/id** - updates a hero **Update**
  - currently only a name change
  - see `putJson` call in `client/src/hero-detail/hero-detail.js`
- **DELETE /hero/id** - deletes a hero **Delete**
  - see `deleteResource` call in `client/src/hero-list/hero-list.js`

We saw these earlier in curl examples.

# Demo Database ...

- Uses MySQL
- To start database daemon

not needed if using Windows and the MySQL server is running as a Windows service

- `cd server`
- `npm run dbstart`
- listens on port 3306 by default
- to stop, `npm run dbstop`

- Database schema is defined in `server/ddl.sql`

```
drop database if exists tour_of_heroes;
create database tour_of_heroes;

use tour_of_heroes;

create table hero (
  id int auto_increment primary key,
  name varchar(100) not null,
  unique uniqueName (name) ← a constraint
);
```

- After changing schema or to restore data

- `cd server`
- `npm run dbsetup` (runs this)
- restart server so it can connect to new database
- refresh browser

```
insert into hero (id, name) values
(11, 'Mr. Nice'),
(12, 'Narco'),
(13, 'Bombasto'),
(14, 'Celeritas'),
(15, 'Magenta'),
(16, 'RubberMan'),
(17, 'Dynamia'),
(18, 'Dr IQ'),
(19, 'Magma'),
(20, 'Tornado');
```



# ... Demo Database

- To examine database
  - `cd server`
  - `npm run dbi`
  - `show tables;`
  - enter SQL queries like `select * from hero;`

```
npm run dbi

> server@1.0.0 dbi /Users/Mark/Documents/programming/languages/javascript/react/react-tour-of-heroes/server
> mysql --user root -p tour_of_heroes

Enter password:
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1507
Server version: 5.7.20 Homebrew

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
mysql> show tables;
+-----+
| Tables_in_tour_of_heroes |
+-----+
| hero                      |
+-----+
1 row in set (0.00 sec)

mysql> select * from hero;
+----+-----+
| id | name      |
+----+-----+
| 13 | Bombasto  |
| 14 | Celeritas |
| 18 | Dr IQ     |
| 17 | Dynama    |
| 19 | Magma     |
| 15 | Magneta   |
| 11 | Mr. Nice  |
| 12 | Narco     |
| 16 | RubberMan |
| 20 | Tornado   |
+----+-----+
10 rows in set (0.00 sec)

mysql>
```

# Hands On Exercise #1

- Add a REST service to the “Tour of Heroes” app
- Implement ability to search for all heroes whose name contains a given string
  - add a route and a middleware function in `server/src/hero-router.js`

```
router.get('/contains/:contains', wrap(filterHeroes));

export async function filterHeroes(req) {
  const {contains} = req.params;
  // Set sql variable to a SQL string
  // that selects all heroes whose names
  // contain the string in the variable contains.
  // Hint: `where name like "%${contains}%"`
  return conn.query(sql);
}
```

solution is in `server/src/filter-heroes.solution`

- Test using curl or Postman
  - send GET request to proper URL



# Hands On Exercise #2

for an example of using **JWT** and cookies, see <https://github.com/mvolkmann/node-rest-demo>

- Implement middleware that performs authentication

- modify `server/src/index.js`
- get "token" from `Authorization` header
- verify it is set to 'magic token'
- if HTTP method is `OPTIONS`, don't check

```
function authenticate(req, res, next) {  
  // Don't require authentication for OPTIONS requests.  
  // Determine this by checking req.method.  
  // Use req.get(name) to get value of a request header.  
  // If bad token, set response status to 401  
  // and send the text "Unauthorized".  
  // If good token, call next().  
}  
  
// Authenticate only routes that start with /hero.  
app.use('/hero', authenticate);
```

solution is in `server/src/authenticate.solution`

- Test using curl or Postman

- GET requests to `localhost:3001/hero` should fail unless required `Authorization` header set to "magic token" is present

In a real app, could have a login page that

- sends username and password to server
- gets an encrypted token that contains things like username, client IP address, and expiration timestamp
- includes encrypted token in `Authorization` request header of all REST calls

The server could decrypt tokens and produce an error if the token isn't recognized or has expired.