KINGSLAND UNIVERSITY

Workshop

REST API

Building a Simple REST API

Table of Contents

{REST API}

REST and RESTful Service

Dividing Client and Server

REST and RESTful Services

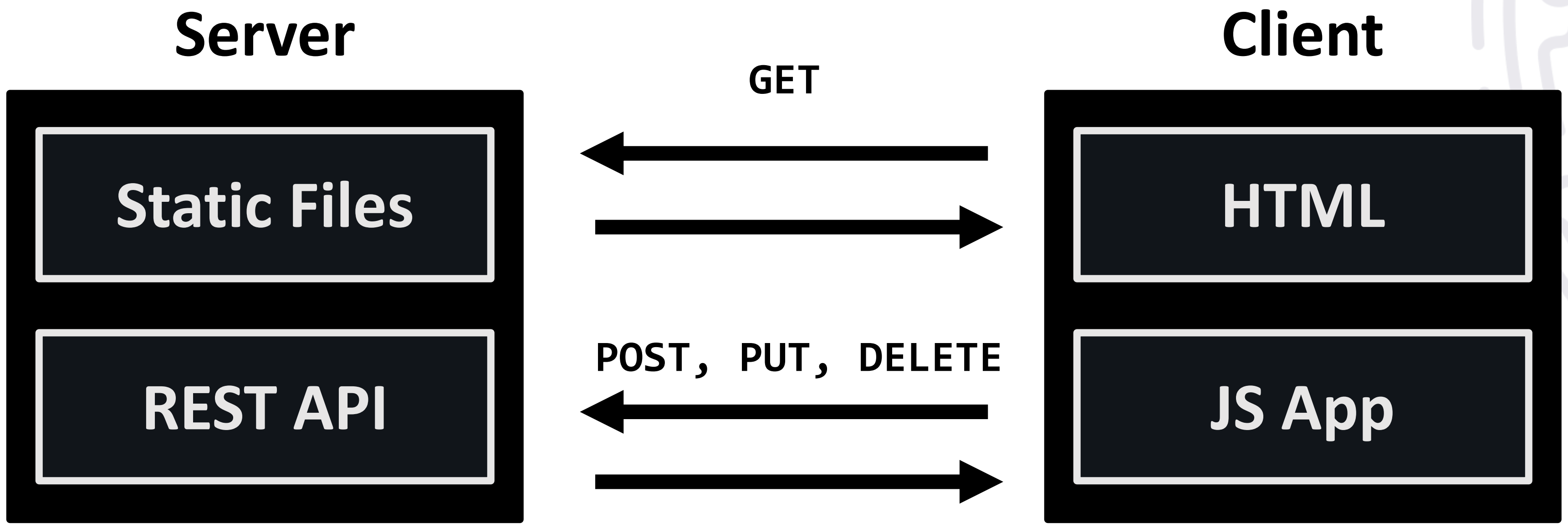
- **Re**presentational **S**tate **T**ransfer (REST)
  - Architecture for client-server communication over HTTP
  - Resources have **URI** (address)
  - Can be created / retrieved / modified / deleted / etc...
- RESTful API / RESTful Service
  - Provides access to server-side resources via HTTP and REST

REST and RESTful Services – Example

- Create a new post
  - **POST** http://some-service.org/api/posts
- Get all posts / specific post
  - **GET** http://some-service.org/api/posts
  - **GET** http://some-service.org/api/posts/17
- Delete existing post
  - **DELETE** http://some-service.org/api/posts/17
- Replace / modify existing post
  - **PUT / PATCH** http://some-service.org/api/posts/17

REST Services with Express

- Websites that use **REST services** are more **interactive**
  - The client can make **AJAX requests** without refreshing the page
  - Necessary for **Single Page Application** (e.g. using React, Angular, Vue.js)



EXPRESS

REST API with Express.js

Initial Configurations

Installing Packages

- Install the following packages

```
npm i –E body-parser
```

```
npm i –E express
```

```
npm i –E express-validator
```

```
npm i –E jsonwebtoken
```

```
npm i –E mongoose
```

Initial Middleware & Config

- Requesting data in JSON format

```
app.use(bodyParser.json())
```

- Setting up router modules

```
app.use('/feed', feedRoutes)
app.use('/auth', authRoutes)
```

- Creating an **express app** and listening to a port

```
app.listen(port, () => {
  console.log(`REST API   listening on port: ${port}`) })
```

Setting Up Router Module

- Using the Express.js Router

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

router.get('/posts', feedController.getPosts);
router.post('/post', feedController.createPost);
router.delete('/post/:postId', feedController.deletePost);
router.get('/post/:postId', feedController.getPostById);
router.put('/post/:postId', feedController.updatePost);

module.exports = router;
```

Fetching Data Example (GET)

- Fetching Data in **JSON** format and returning

```
getPosts: (req, res) => {
    Post.find()
      .then((posts) => {
        res
          .status(200)
          .json({ message: 'Fetched posts successfully.', posts });
      })
      .catch((err) => {
        res.status(500)
          .json({ message: 'Server error!'})
      });
}
```

Creating Data Example (POST)

- Persisting into a DB

```
const { title, content } = req.body;
    // Validate data before persisting
    const post = new Post({ title, content });
    post.save()
      .then(() => {
        res.status(201)
          .json({ message: 'Post created successfully!',
            post: post
          })
      })
      .catch((error) => // Handle error }
```

Always return correct status codes!

Live Demo

Setup Express.js REST API

CORS

Cross Origin Resource Sharing

CORS Definition

- Browser security prevents a web page from making requests to a **different domain**
  - This restriction is called **S**ame-**O**rigin **P**olicy (**SOP**)
  - This policy also prevents malicious sites from reading data from your site
- Sometimes you might want to **allow other sites** to bypass this restriction
  - This is where CORS comes to the rescue

Different Origin

- **CORS** is a **W3C** standard that allows a server to "relax" the **SOP**
  - Using **CORS**, a server can **explicitly** allow some cross-origin requests
  - That doesn’t mean all cross-origin requests will be allowed
- Two URLs have the **same origin** if they have
  - Identical **Schemes**, **Hosts** and **Ports** (RFC 6454)

Same vs Different Origin URLs

- Same-origin URLs

```
https://example.com/foo.html
```

```
https://example.com/moo.html
```

```
https://example.com/boo.html
```

- Different-origin URLs

```
https://example.net
```

```
https://www.example.com/foo.html
```

```
http://example.com/foo.html
```

```
https://example.com:9000/foo.html
```

Setting Up CORS in Express.js

- Define **middleware** that sets additional **headers**

```
app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*');

  res.setHeader('Access-Control-Allow-Methods',
   'OPTIONS, GET, POST, PUT, PATCH, DELETE');

  res.setHeader('Access-Control-Allow-Headers',
   'Content-Type, Authorization');

  next();
});
```

Authentication with JWT

Signing and Verifying Tokens

JSON Web Tokens

- **JWT** is a method for representing claims between two parties
  - An open, industry standard – RFC 7519
  - Easy to use, and at the same time – absolutely secured
- When the user successfully **authenticates** (login) using their credentials:
  - A **JSON Web Token** is generated and returned
  - It must be stored (in **local** / **session** storage, **cookies** are also an option)
- Whenever a protected route is accessed, the user agent sends the **JWT**
  - Typically in an **Authorization** header, using the **Bearer** schema

JSON Web Tokens

- **JWT** is **stateless**, nothing is stored on the server
- Here is an example of an encoded and decoded **JSON Web Token**

The parts of the token are separated by dots

As any normal auth JWT also has an expiration

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI
6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDI
yfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6y
JV_adQssw5c
```

The parts of the token are in a strict order

The token data does not change the token format

Decoded

Header: (algorithm, token type)

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload: (data)

```
{ "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Verify Signature

```
HMACSHA256(base64UrlEncode(H...) +
"." + base64UrlEncode(P...), key)
```

Using JWT to Sign Users in

```
signIn: (req, res) => {
  User.findOne({ email: email })
      .then((user) => {
        // Check if user exists
         // Check if the password is correct
        const token = jwt.sign({
          email: user.email,
          userId: user._id.toString()
        }, 'somesupersecret', { expiresIn: '1h' });

         res.status(200).json(
           { message: 'User successfully logged in!',
             token,
             userId: user._id.toString()
           });
      })
      .catch(...)
}
```

Token will expire in one hour

Setting Up Middleware for Authentication

- Accessing specific routes that require **authentication** should sent **authorization headers** with the request in format:
  - Authorization: **Bearer {jwtToken}**

```
const authHeaders = req.get('Authorization');
if (!authHeaders) {
  return res.status(401)
      .json({ message: 'Not authenticated.' })
}
```

```
const token = req.get('Authorization').split(' ')[1];
```

Verifying Token

- We then try and verify our token

```
let decodedToken;
try {
  decodedToken = jwt.verify(token, 'somesupersecret')
} catch(error) {
   return res.status(401)
      .json({ message: 'Token is invalid.', error });
}

req.userId = decodedToken.userId;
next();
```

The same secret we used when signing in

The userId can be used later for verification

Use Middleware with Routing

- Attach the created middleware to every route that **needs** authentication

```
const isAuth = require('../middleware/is-auth');

router.get('/posts', isAuth, …);
router.post('/post', isAuth , …);
router.delete('/post/:id', isAuth, …);
router.get('/post/:id', isAuth);
router.put('/post/:id', isAuth, …);
```

Error Handling and Validation

Using Express-validator

Generic Error Handling Middleware

- When an error occurs it is always good idea to have general **error handling** functionality

```
app.use((error, req, res, next) => {
  const status = error.statusCode || 500;
  const message = error.message;
  res.status(status).json({ message: message });
  next();
});
```

Throwing Custom Errors Example

- Create errors and attach a given status code to that error

```
Post.findById(postId)
    .then((post) => {

        if (!post) {
          const error = new Error('Post not found!');
          error.statusCode = 404;
          throw error;
        }

      // Check if post the current user is the author
      // If not throw 403 error
      Post.findByIdAndDelete(postId);
})
```

Catching Errors

- When the custom error is thrown, we catch it inside the promise rejection

```
Post.findById(postId)
  .then((post) => {
    // Delete post
  })
  .catch(error => {
    if (!error.statusCode) {
        error.statusCode = 500;
    }
    next(error);
  })
```

If there is no status code attached, then something went wrong with the server

The error is sent to the middleware

Using Express-validator

- Express-validator is a set of express.js middleware's
- We define validations **before** a controller action is called

```
const { body } = require('express-validator/check')

router.post('/post/create', isAuth , [
  body('title')
    .trim()
    .isLength({ min: 5 }),
  body('content')
    .trim()
    .isLength({ min: 5 })
], feedController.createPost)
```

Sending Validation Messages to the Client

- To validate an entity call a function that checks the **request body** for errors and adds them in an **array**

```
const { validationResult } = require('express-validator/check');

function validatePost(req, res) {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
     res.status(422).json({
      message: 'Validation failed, entered data is incorrect',
      errors: errors.array()
    });
  } else {
     return true;
  }
}
```

Creating Custom Validations

- Express-validators allows us to create **custom validations** and that send **custom messages**

```
body('email')
     .isEmail()
     .withMessage('Please enter a valid email.')
     .custom((value, { req }) => {
       return User.findOne({ email: value }).then(userDoc => {
         if (userDoc) {
           return Promise.reject('E-Mail address already exists!');
         }
       })
     })
```

- More here: https://express-validator.github.io/docs/

Summary

- **REST** is an architecture for client-server communication over HTTP
- Building a **RESTful service** in Express.js
- Using **CORS**, a server can **explicitly** allow some cross-origin requests
- **JWT** is a method for representing claims between two parties

Questions?

License

KINGSLAND UNIVERSITY

THANK YOU