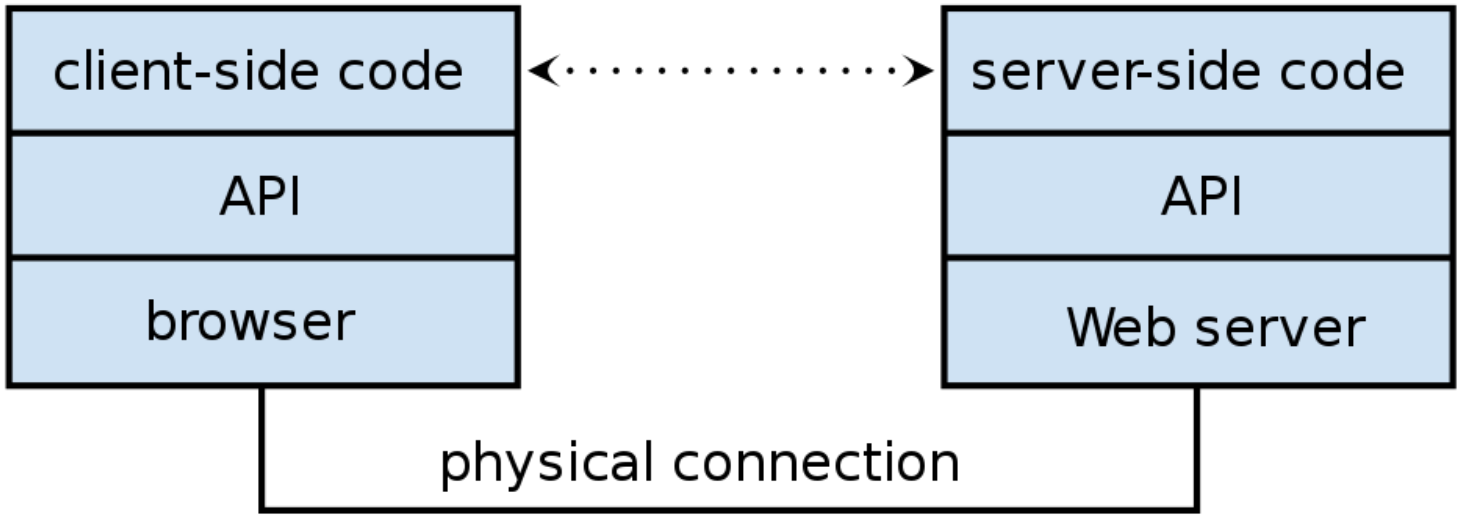


Web API Design

Frank Walsh

Agenda

- REST
- API Value
- API Design
- Express Middleware
- Routing in Express
- The Request and Response object

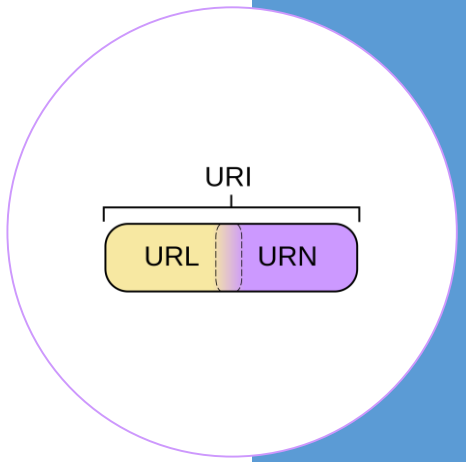
REST

- Short for Representational State Transfer
- Set of Principles for how web should be used
- Coined by Roy Fielding
    - One of the HTTP creator

•A set of principles that define how Web standards(HTTP and URIs) can be used.

Key REST Principles

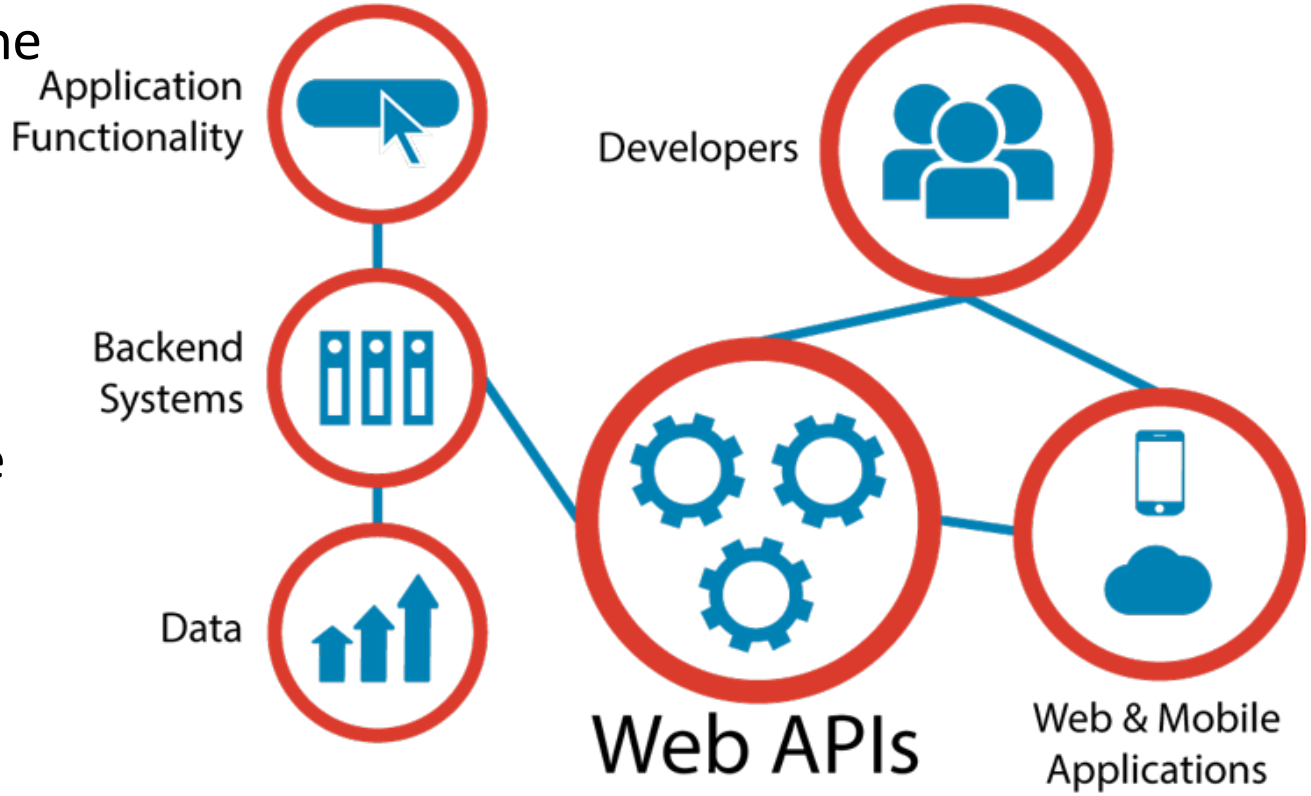
1. Every “thing” has an identity
   - URL
2. Link things together
   - Hypermedia/Hyperlinks
3. Use standard set of methods
   - HTTP GET/POST/PUT/DELETE
   - Manipulate resources through their representations
4. Resources can have multiple representations
   - JSON/XML/png/...
5. Communicate stateless
   - Should **not** depend on server state.



Web APIs

Web APIs

- Programmatic interface exposed via the web
- Uses open standards typically with request-response messaging.
  - E.g messages in JSON or XML
  - HTTP as transport
  - URIs
- Example would be Restful web service described in previous lectures.
- Typical use:
  - Expose application functionality via the web
  - Machine to machine communication
  - Distributed systems



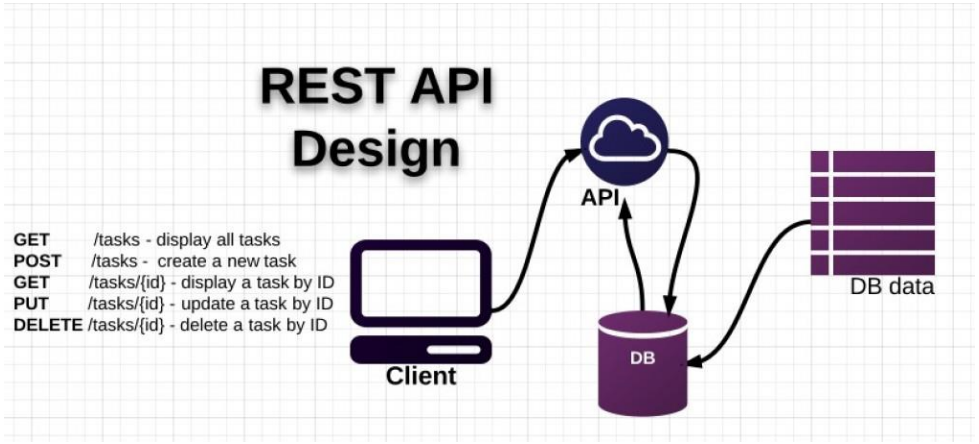
“API First” approach

- Collaboratively design, mockup, implement and document an API **before** the application or other channels that will use it even exist.
- Uses “clean-room” approach.
  - the API is designed with little consideration for the existing IT estate.
  - the API is designed as though there are no constraints.

source: http://www.programmableweb.com/news/introduction-to-api-first-design/analysis/2016/10/31

Traditional API Design

- API design happens after the release of some a data-rich application
  - Existing application “wrapped” in API
- Created as an afterthought.
  - Tightly bound application needs data/function exposed as API.
  - Shoe-horned in as a separate entity.



Advantages of API First

- Suits multi-device environment of today.
- An API layer can serve multiple channels/devices.
  - Mobile/tablet/IoT device
- Scalable, modular, cohesive and composeable
  - If designed properly(e.g. microservice architecture)
  - See later slides
- Concentrate on function first rather tha data

APIs in the Internet of Things

- Many new IoT devices being released.
- Devices are limited on their own
  - It's the innovative use of those devices with accompanying APIs that generate value
- "Build a better mousetrap, and the world will beat a path to your door" - Ralph Waldo Emerson
  - Rentokil believe they have using APIs(https://www.computerworlduk.com/it-business/rentokil-on-iot-rat-traps-cash-for-apps-incentives-apis-3612866/)
  - Rentokil increased operational efficiency through the automatic notifications of a caught animal and its size

API Design

- Use principle of developer-first
  - put target developers' interests ahead of other considerations
  - Strive for a better developer experience
- Commit to RESTful APIs
- Use a Interface Description Language like:
  - RESTful API Markup Language (RAML)
  - Swagger
- Take a grammatical approach to the functionality
- Keep interface simple and intuitive

API Design

- In Rest, everything is based around resources
  - the “things” you’re working with are modelled as resources described by URI paths--like /users, /groups, /dogs
  - Notice they are **nouns .**
  - **Verbs in URLs are BAD**
- The things that you do on these things (or nouns) are characterised by the fixed set of HTTP methods
  - What GET,POST,PUT does is something that the designer/developer gets to put into the model.
- The metadata (the adjectives) is usually encoded in HTTP headers, although sometimes in the payload.
- The responses are the pre-established HTTP status codes and body. (200, 404, 500 etc.)
- The representations of the resource are found inside the body of the request and respons

Resource	POST create	GET read	PUT update	DELETE delete
/dogs	Create a new dog	List dogs	Bulk update dogs	Delete all dogs
/dogs/1234	Error	Show Bo	If exists update Bo If not error	Delete Bo

API Design - Containment

- URIs embed ids of “child” resources
- Post creates child resources
- Put/Delete for updating /removing resources

Resource	GET	POST	PUT	DELETE
/api/posts	get all posts	add a post	N/A	N/A
/api/posts/:postId	get a post	N/A	update a post	N/A
/api/posts/:postId/upvotes	N/A	upvote a post	N/A	N/A
/api/posts/:postId/comments	Get comments for post	Add a comment	N/A	N/A
/api/posts/:postId/comments/:commentId/upvotes	get upvotes	upvotes a comment	N/A	N/A

Express Middleware

- Express is a Routing and Middleware framework.
  - You've seen the routing in the previous lab
- Middleware functions have access to the Request,Response and the **next()** function
  - The next function calls the next middleware function.
- Use middleware to
  - Change the request/response
  - End the request/response cycle
  - Call the next middleware in the stack.
- If middleware does not call next() or return, express will just hang
  - Can be an issue with promises but can be resolved

```
11  const middleware1 = (req, res, next) => {
12    console.log('in middleware 1');
13    next();
14  };
15
16  app.use(middleware1);
17  app.use(express.static('public'));
```

Express Middleware Types

- 3rd Party (e.g. body-parser)
- Router level (more later)
- App level (app.use(…) in previous slide)
  - Every request is handled
- Error handlers
  - Takes error as first parameter
    (err,req,res,next) => { …. }
- Baked in
  - Express.static()

Express Middleware – Error Middleware

```
11  const middleware1 = (req, res, next) => {
12    console.log('in middleware 1');
13    next(new Error('BOOM!')); // for error handler example
14    // next(); // for general middleware example
15  };
16
17  const errorHandler1 = (err, req, res, next) => {
18    console.log('error handler!!!');
19    console.log(err);
20    next();
21  };
22
23  app.use(middleware1);
24  app.use(express.static('public'));
25  app.use('/api/contacts', contactsRouter);
26  app.use(errorHandler1);
```

Raise error and pass on to next error handling middleware in middleware stack

NOTE: Middleware stack processed in the order it appears in script.

Express Routers

- Can have several "routers" to implement your APIs.
- Router can have its own routing and middleware
  - Good for multiple APIs/ versioning
- Still uses the application level middleware of express app.

Exports router instance

/api/contacts/index.js (contacts router)

```
import express from 'express';
import {contacts} from './contacts';

const router = express.Router(); // eslint-disable-line
router.get('/', (req, res) => {
  res.send({contacts: contacts});
});

export default router;
```

/index.js (express app)

```
import dotenv from 'dotenv';
import express from 'express';
import contactsRouter from './api/contacts';

dotenv.config();

const app = express();


app.use(express.static('public'));
app.use('/api/contacts', contactsRouter);
app.use(errorHandler1);
```

Mount router to URL. **/api/contacts** becomes **Base Route** for router

Express Request Object

- The **req** object represents the HTTP request.
  - by convention, the object is referred to as '**req**', Response is '**res**'
- Can use it to access the request query string, parameters, body, HTTP headers.
- Example:

Parameterised URL. Access using req.params.id

```
router.get('/user/:id',(req, res)=>{
  res.send('user ' + req.params.id);
});
```

Express Request Object
req.body

- Contains data submitted in the request body.
- Need body-parsing middleware such as **body-parser**.
- This example shows how to use body-parsing middleware to populate req.body.

```
const server = express();

//configure body-parser middleware
server.use(body_parser.json());
//parses application/x-www-form-urlencoded
server.use(body_parser.urlencoded());
...
router.post('/echo',(req, res)=>{
  console.log(req.body);
  res.json(req.body);
});
```

Express Response Object

- The res object represents the HTTP response that an Express app sends when it gets an HTTP request.

```
//Add a contact
router.post('/', (req, res) => {
    let newContact = req.body;
    if (newContact){
          contacts.push({name: newContact.name, address : newContact.address});
          res.status(201).send({message: "Contact Created"});
      }else{
          res.status(400).send({message: "Unable to find Contact"});
      }
});
```

Response Properties

- **res.send([body])**
  - The body parameter can be a String, an object, or an Array.
  - For example:

```
res.send({ some: 'json' });
res.send('<p>some html</p>'); res.status(404).send('Sorry, we cannot find that!');
res.status(500).send({ error: 'something blew up' });
```

Response Properties

- **res.json([body])**
  - –Sends a JSON response. This method is identical to res.send() with an object or array as the parameter.

```
res.json({ user: 'tobi' })
res.status(500).json({ error: 'message' })
```

Response Properties

- **res.format(object)**
  - –Performs content-negotiation on the Accept HTTP header on the request object
  - –Addresses "multiple representations" REST principle

```
res.format({
  'text/plain': function(){
    res.send('hey');
  },

  'text/html': function(){
    res.send('<p>hey</p>');
  },

  'application/json': function(){
    res.send({ message: 'hey' });
  },

  'default': function() {
    // log the request and respond with 406
    res.status(406).send('Not Acceptable');
  }
});
```

Filters

If you want to authenticate for access to resources you can use multiple callbacks built into express routing

Multiple Callbacks

```
function requireLogin(req, res, next) {
  if (req.session.loggedIn) {
    next(); // allow the next route to run
  } else {
    // require the user to log in
    res.redirect("/login"); // or render a form, etc.
  }
}

// Automatically apply the `requireLogin` middleware to all
// routes starting with `/admin`
router.all("/admin/*", requireLogin, (req, res, next)=> {
  next(); // if the middleware allowed us to get here,
          // just move on to the next route handler
});

router.get("/admin/posts", (req, res)=> {
  // if we got here, the `app.all` call above has already
  // ensured that the user is logged in
});
```

Middleware with Async await/promises

- Express will not detect rejected promise automatically
  - Error handling middleware will not be called – causes app to hang.
- Couple of ways to address this
  - Use try/catch in each async function/promise (lots of repetitive code)
  - Use a helper function that wraps our express routes to handle rejected promises.

```
1  const asyncMiddleware = fn =>
2    (req, res, next) => {
3      Promise.resolve(fn(req, res, next))
4        .catch(next);
5    };
```

- Handy: someone has published a NPM package:
  npm install --save express-async-handler

Further Reference

- ExpressJS.com - Official Express Homepage
- Node and Express Tutorial