

# Node Program

## Express.js



Node.js version: 5.1

Last updated: Jan 2016

# Express

Express is the most popular web application framework for Node. It is easy to work with as it ties into Node's functional paradigm.

- Deliver static content (or consider using nginx)
- Modularize business logic
- Construct an API
- Connect to various data sources

# DEMO

Core http module API: <http://bit.ly/1StXFsG>



With Express you can develop APIs  
faster!

## Express vs. http

- URL params and query strings parsing
- Automatic response headers
- Routes and better code organization
- Myriads of plugins (called middleware)
- Request body parsing (with a module)
- Authentication, validation, session and more! (with modules)

# Installing Dependency

```
$ npm install express --save
```

```
$ npm install express@4.13.3 --save
```

# Installing Scaffolding

Install Express.js command-line generator:

```
$ npm install -g express-generator
```

## Using the Generator

```
$ express todo-list-app  
$ cd todo-list-app  
$ npm install  
$ node app
```



# Structure

- `app.js`: main file, houses the embedded server and application logic
- `/public`: contains static files to be served by the embedded server
- `/routes`: houses custom routing for the embedded server
- `/views`: contains templates that can be processed by a template engine

# app.js

1. Imports and instantiations
2. Configurations
3. Middleware
4. Routes
5. Bootup

# Configuring Express

The Express server needs to be configured before it can start

Manage configuration via the `set` method:

```
var express = require('express')
var app = express()
app.set('port', process.env.PORT || 3000)
app.set('views', 'templates') // The directory the templates are stored in
app.set('view engine', 'jade')
```

# Node.js Middleware Pattern

# What is Middleware

Middleware pattern is a series of processing units connected together, where the output of one unit is the input for the next one. In Node.js, this often means a series of functions in the form:

```
function(args, next) {  
  // ... Run some code  
  next(output) // Error or real output  
}
```

# Continuity

Request is coming from a client and response is sent back to the client.

`request->middleware1->middleware2->...middlewareN->route->response`

## Organizing Code

database in `app.js`, but we need it in `routes/users.js` where our `/users` routes are located

How to pass the database reference? Something like this?

```
var users = require('./routes/users.js')(database)
```

There is a better way!

# Connect Framework

Express leverages the Connect framework to provide the middleware functionality. Middleware are used to manage how a request should be handled.



# Applying Connect/Express Middleware

Example:

```
var express = require('express')
var app = express()
//... Define middleware1-N
app.use(middleware1)
app.use(middleware2)
...
app.use(middlewareN)
...
```

# Middleware Order

Middleware are executed in the order specified:

```
var logger = require('morgan')
var bodyParser = require('body-parser')
...
app.use(logger('dev'))
app.use(bodyParser.json())
```

# Two Categories of Express Middleware

1. npm modules, e.g., body-parser
2. Custom middleware

# Creating Middleware

Custom middleware is easy to create with a reference:

```
var middleware = function (request, response, next) {  
  // Modify request or response  
  // Execute the callback when done  
  next()  
}  
app.use(middleware)
```

## Creating Middleware

Or with anonymous function definition:

```
app.use(function (request, response, next) {  
  // Modify request or response  
  // Execute the callback when done  
  next()  
})
```

## Passing References

request is **always** the same object in the lifecycle of a single client request to the Express server

This solves the database reference problem:

```
app.use(function (request, response, next) {  
  request.database = database  
  next()  
})
```

## Most Popular and Useful Connect/Express Middleware

```
$ npm install <package_name> --save
```

- `body-parser` request payload
- `compression` gzip
- `connect-timeout` set request timeout
- `cookie-parser` Cookies
- `cookie-session` Session via Cookies store

## Connect/Express Middleware

- `csrf` CSRF
- `errorhandler` error handler
- `express-session` session via in-memory or other store
- `method-override` HTTP method override
- `morgan` server logs
- `response-time`



# Connect/Express Middleware

- `serve-favicon` favicon
- `serve-index`
- `serve-static` static content
- `vhost`

# Other Popular Middleware

- `cookies` and `keygrip`: analogous to `cookieParser`
- `raw-body`
- `connect-multiparty`, `connect-busboy`
- `qs`: analogous to `query`
- `st`, `connect-static` analogous to `staticCache`

# Other Popular Middleware

- `express-validator`: validation
- `less`: LESS CSS
- `passport`: authentication library
- `helmet`: security headers
- `connect-cors`: CORS
- `connect-redis`

# Template Engine

Setting the `view engine` variable to `jade` for instance, would trigger

the following function call internally

```
app.set('view engine', 'jade') // Shorthand
```

```
// Does the same as the above
```

```
app.engine('jade', require('jade').__express)
```

# Template Engine

Custom callbacks can be defined to parse templates

```
app.engine([format], function (path, options, callback) {  
  // Template parsing logic goes here  
});
```

Note: custom callbacks are useful if the template engine doesn't export an **\_\_express** function

# Express Bootup

```
var http = require('http'),  
    express = require('express')  
  
var app = express()  
  
// ... Configurations, middleware and routes  
  
var server = http.createServer(app)  
server.listen(app.get('port'), function () {  
    // Do something... maybe log some info?  
});
```

## Bootup 2

```
var http = require('http'),  
    express = require('express')  
  
var app = express()  
  
// ... Configurations, middleware and routes  
  
app.listen(app.get('port'), function () {  
    // Do something... maybe log some info?  
});
```

# Launching the App

```
$ node server
```

```
$ nodemon server
```

```
$ node-dev server
```

```
$ forever server
```

```
$ pm2 server
```

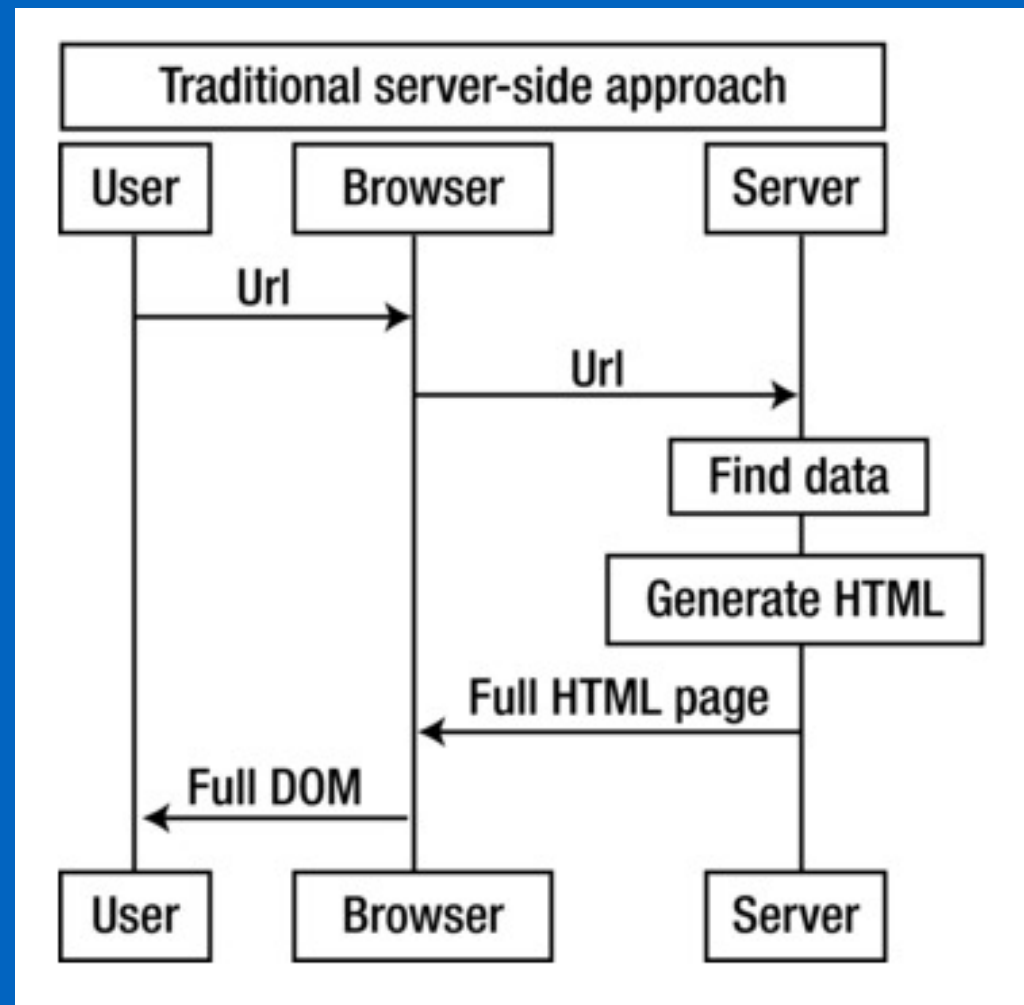


Express is awesome! 🚀

# Building a RESTful API

# Traditional Web App

Also called thick server.

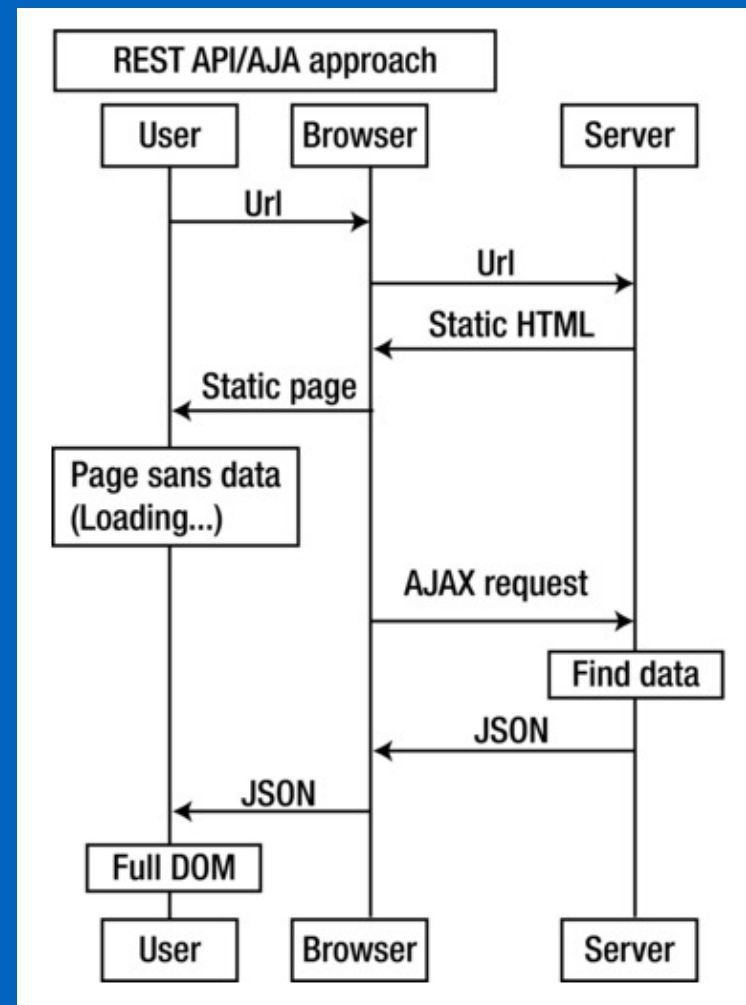


# Traditional Web App Problems

- Slow and single-tasking (not multitasking)
- Poor and unresponsive UX (user experience)
- Duplication of data hogs bandwidth (HTML)

# API + AJAX/XHR Web App

Also called thick client

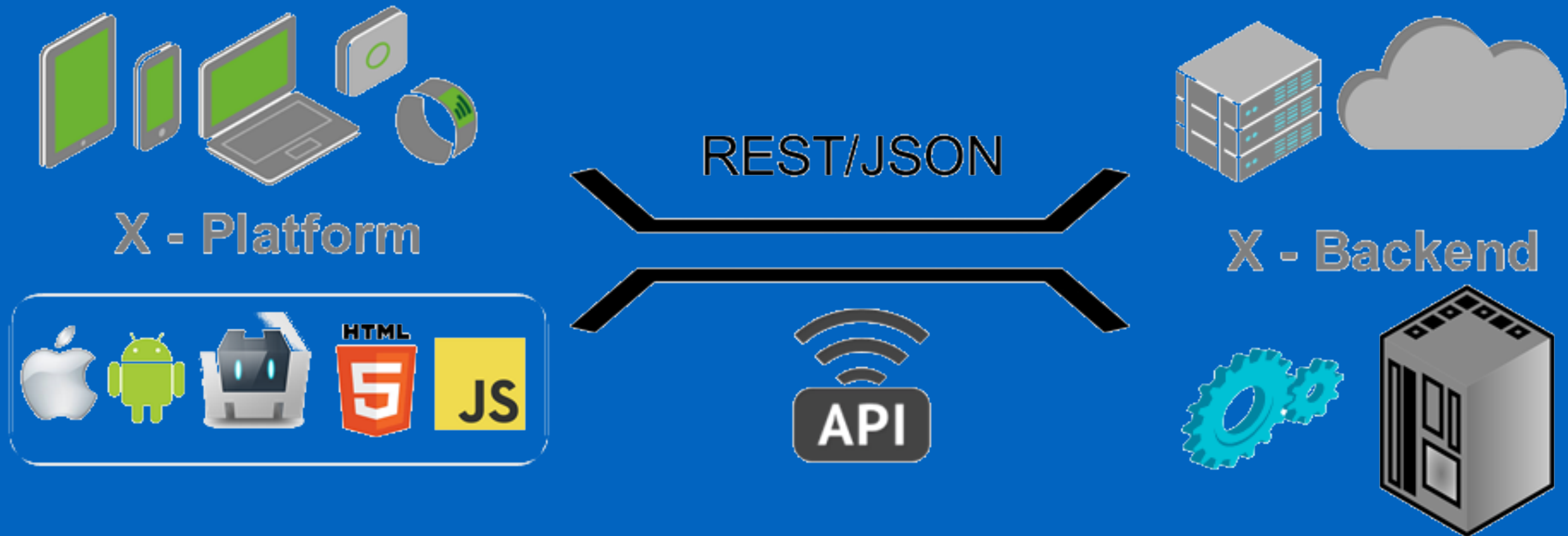


# Advantages of a Thick Client

- Responsive interface and UX
- Only data is transmitted (JSON)
- Re-use of the core functionality
- Asynchronous tasks
- Real-time apps

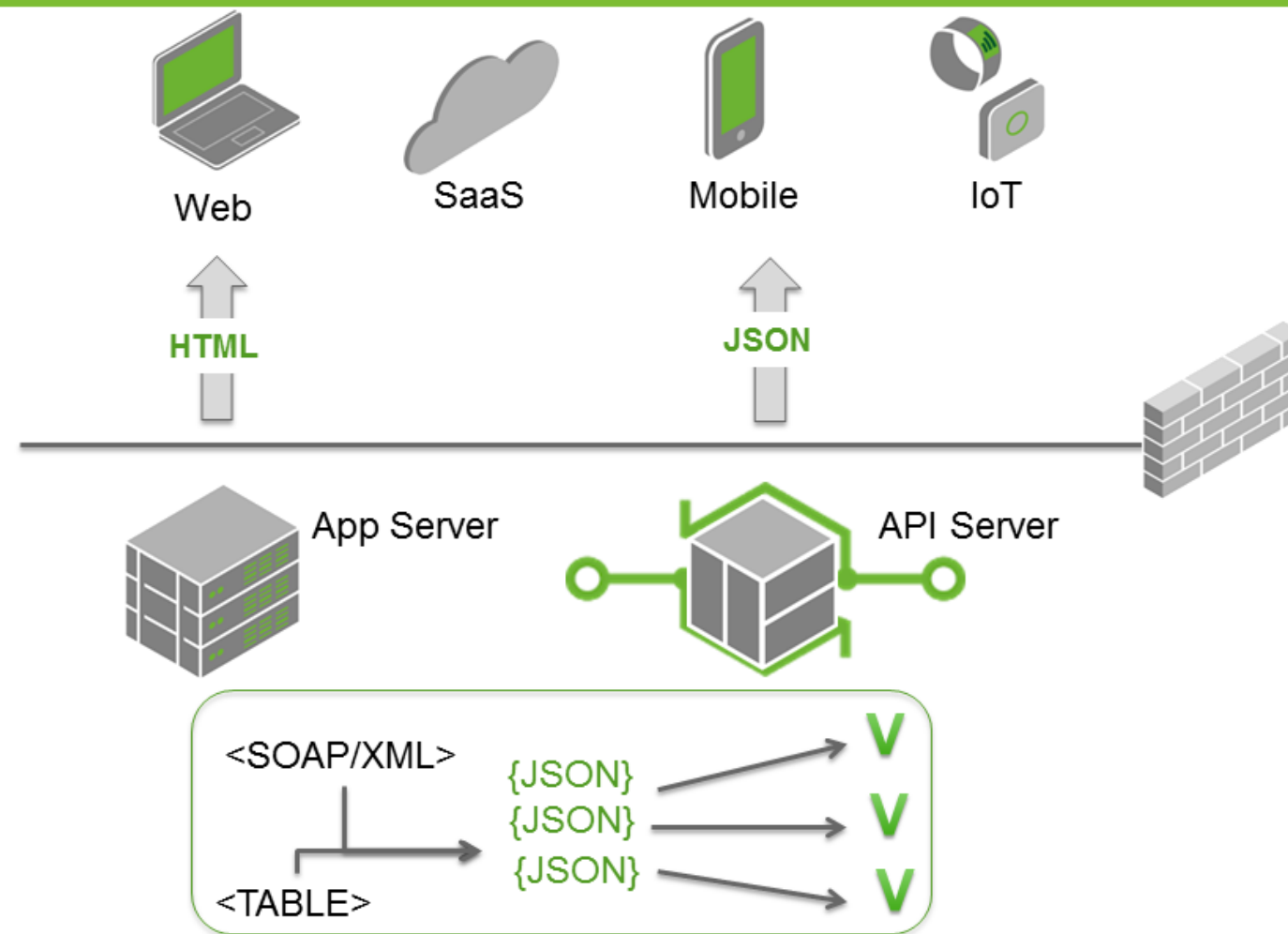
# Node, SPAs and REST

Build an API once and use everywhere



# API Decomposition

API “Decomposition” is the game changer

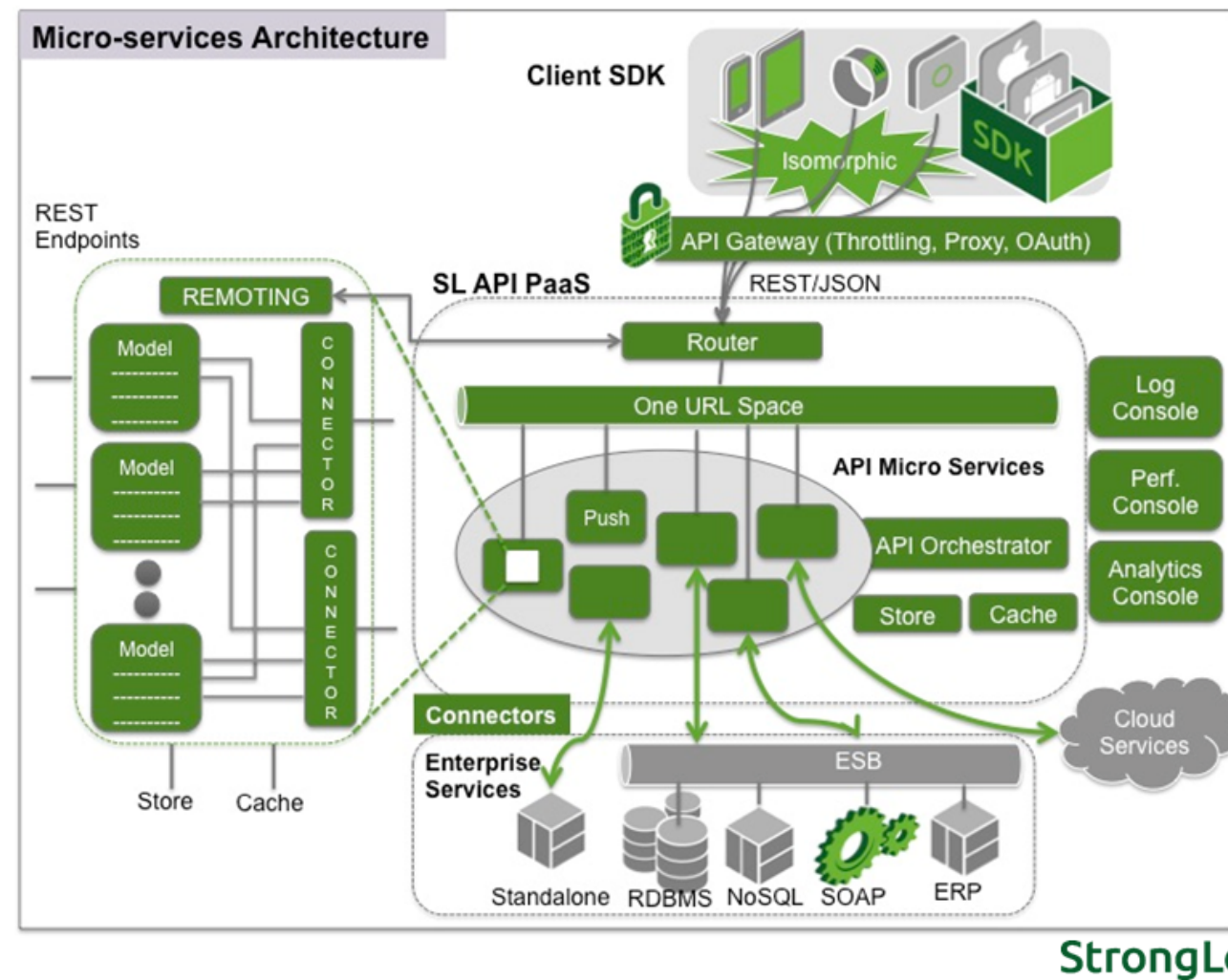


StrongLoop



# Microservices

Micro-services have arrived



# REST Basics

REpresentational State Transfer (REST) is an architectural pattern for developing network applications

REST systems aim to keep things simple when connecting to and exchanging data between machines

# Why HTTP?

HTTP is the ideal protocol for REST, given its stateless nature and client-server architecture

- REST is far simpler compared to Remote Procedure Calls (RPC) and Web Services (SOAP, UDDI, etc)
- RPCs and Web services rely on complex vocabularies for communication
- Each new operation is a new vocabulary entry, increasing code complexity

# REST Verbs

REST uses HTTP requests (and verbs) for CRUD operations

- GET
- PUT
- POST
- DELETE

# REST Verbs

And sometimes...

- PATCH
- HEAD
- OPTIONS

# Common Endpoints

GET	/tickets	- Retrieve a list of tickets
GET	/tickets/12	- Retrieve a specific ticket
POST	/tickets	- Create a new ticket
PUT	/tickets/12	- Update ticket #12
DELETE	/tickets/12	- Delete ticket #12
PATCH	/tickets/12	- Partially update ticket #12
OPTIONS	/tickets/12	- What can I do to ticket #12?
HEAD	/tickets/12	- What headers would I get if I tried to get ticket #12?

# "Resources"

Resources are entities that can be stored on a computer, such as:

- Files
- Database entries
- Processed output from functions

## "Resources"

REST uses HTTP requests and responses to provide **representations** of resources

For example, the current version of a file available for download via its URL is a representation of a file resource

Modifying a resource, such as changing the contents of a file or deleting it, is also a resource state that can be represented via requests and responses in a REST system



# REST API Examples

# Handlers Signatures

- `function(request, response, next) {}`: request handler signature
- `function(error, request, response, next) {}`: error handler signature

## GET Route

```
app.get('/users', function (request, response) {  
  // Code to retrieve users  
  response.send(user)  
})
```

## Accessing URL Parameters

A URI segment can be parameterized by prefixing it with a semi-colon

```
app.get('/users/:id/:another/:segment', function (request, response) { ... })
```

These dynamic parameters can then be accessed via the request's **params** object

GET /users/:id

```
request.params.id
```

# Multiple URL Parameters

GET /users/:id/:some/:filter

`request.params.id`

`request.params.some`

`request.params.filter`

# GET

To allow retrieval by id...

```
app.get('/users/:id', function (request, response) {  
  var id = request.params.id  
  // Code to retrieve a single user  
  response.send(user)  
})
```

# GET

GET handlers can also be used to retrieve a collection of resources

```
app.get('/users', function (request, response) {  
  // Code to retrieve multiple users  
  response.send(users)  
})
```

# POST

To create a resource...

```
app.post('/users', function (request, response) {  
  var username = request.body.username  
  var email = request.body.email  
  // ...  
  // Code to create a new user  
  response.send(user)  
});
```

Or maybe just send back the endpoint to get the user..

```
response.send('/api/user/' + user.id)
```



# PUT

To update a resource (or create if it doesn't exist, perhaps)...

```
app.put('/users/:id', function (request, response) {  
  var id = request.params.id  
  // Check if the user exists  
  ...  
  if (exists) {  
    // Code to modify the user  
  } else {  
    // Code to create the user  
  }  
  response.send(user);  
});
```

# DELETE

To delete a resource, create a DELETE handler for the desired URI

```
app.delete('/users/:id', function (request, response) {  
  var id = request.params.id;  
  // code to delete the user  
  response.send(user); // or maybe the URL to create a new user?  
});
```

Note: `del` is deprecated.

# HTTP Requests

A client's HTTP request is accessible from within routing handlers

It is the first argument in the handler's callback

```
app.get('/users/:id', function (request, response) {  
  // 'req' is the enhanced http request object  
});
```

Note: access to the request object grants insight into the client's HTTP request, providing data on the request header, body, et al.

# Query Strings

Express converts a URL's query string into JSON

It can be accessed via the request's **query** object

```
GET http://localhost:3000/?name=Bruce+Wayne&age=40&occupation=Batman
```

```
request.query.name // "Bruce Wayne"
```

```
request.query.age // "40"
```

```
request.query.occupation // "Batman"
```

## Request Body

Enable the `json()` and `urlencoded()` middleware to convert raw form data into JSON

```
$ npm install body-parser --save
```

## Parsing Request Body

Import middleware:

```
var bodyParser = require('body-parser')
```

Parse application/json

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

Usage: single-page applications and other JSON REST clients.

## Parsing Request Body

Parse `application/x-www-form-urlencoded`

```
app.use(bodyParser.urlencoded({extended: false}))
```

Usage: web forms with `action` attribute.

## Accessing Form Data

Form data is then accessible via the request's **body** object (ulrencoded)

```
// POST name=Bruce+Wayne&age=40&occupation=Your+Average+Businessman
```

```
request.body.name
```

```
request.body.age
```

```
request.body.occupation
```



# File Uploads

File uploads from web forms (multipart/form-data) can be parsed with these libraries:

- <https://github.com/expressjs/multer>
- <https://github.com/yahoo/express-busboy>
- <https://github.com/mscdex/connect-busboy>
- <https://github.com/andrewrk/node-multipart>

# Parsing JSON

Parse various different custom JSON types as JSON

```
app.use(bodyParser.json({ type: 'application/*+json' })))
```

# Parsing Buffer

Parse some custom thing into a Buffer

```
app.use(bodyParser.raw({ type: 'application/vnd.custom-type' })))
```

# Parsing HTML

Parse an HTML body into a string

```
app.use(bodyParser.text({ type: 'text/html' }))
```

## HTTP Verbs and Routes

- `app.get(urlPattern, requestHandler[, requestHandler2, ...])`
- `app.post(urlPattern, requestHandler[, requestHandler2, ...])`
- `app.put(urlPattern, requestHandler[, requestHandler2, ...])`
- `app.delete(urlPattern, requestHandler[, requestHandler2, ...])`

## HTTP Verbs and Routes

- `app.all(urlPattern, requestHandler[, requestHandler2, ...])`
- `app.param([name,] callback):`
- `app.use([urlPattern,] requestHandler[, requestHandler2, ...])`

# Request

- `request.params`: parameters middleware
- `request.param`: extract one parameter
- `request.query`: extract query string parameter
- `request.route`: return route string

# Request

- `request.cookies`: cookies, requires `cookieParser`
- `request.signedCookies`: signed cookies, requires `cookie-parser`
- `request.body`: payload, requires `body-parser`



## Request Header Shortcuts

- `request.get(headerKey)`: value for the header key
- `request.accepts(type)`: checks if the type is accepted
- `request.acceptsLanguage(language)`: checks language
- `request.acceptsCharset(charset)`: checks charset
- `request.is(type)`: checks the type
- `request.ip`: IP address

## Request Header Shortcuts

- `request.ips`: IP addresses (with trust-proxy on)
- `request.path`: URL path
- `request.host`: host without port number
- `request.fresh`: checks freshness
- `request.stale`: checks staleness
- `request.xhr`: true for AJAX-y requests

## Request Header Shortcuts

- `request.protocol`: returns HTTP protocol
- `request.secure`: checks if protocol is https
- `request.subdomains`: array of subdomains
- `request.originalUrl`: original URL

# HTTP Responses

The response object is also accessible via routing handlers in Express

It is the second argument in the handler's callback

```
app.get('/users/:id', function (request, response) {  
  // 'response' is the enhanced response from http  
})
```

The response object can be used to modify an HTTP response before sending it out

## Express Response Method

- `response.redirect(status, url):` redirect request
- `response.send(status, data):` send response
- `response.json(status, data):` send JSON and force proper headers

## Express Response Method

- `response.sendFile(path, options, callback):` send a file
- `response.render(templateName, locals, callback):` render a template
- `response.locals:` pass data to template

# HTTP Status Codes

To specify a status code, use the response object's **status** function

```
app.get('/user/:id', function (request, response) {  
  // Logic to check for user  
  if (!exists) {  
    response.status(404)  
  } else if (authorized) {  
    response.status(200)  
  } else {  
    response.status(401)  
  }  
  // ...  
});
```

# HTTP Status Codes

- 2XX: for successfully processed requests
- 3XX: for redirections or cache information
- 4XX: for client-side errors
- 5XX: for server-side errors

Note: for 3xx status codes, the client must take additional action following the completion of the current request



## Sending a Response

Use the response object's **send** function to send the client a response

```
app.get('...', function (request, response) {  
  response.send('Hello World!')  
})
```

# Sending a Response

The content-type is determined given the type of argument passed

```
response.send('Hello World!')           // Content-type: text/plain
response.send([ 5, 7, 9 ])               // Content-type: application/json
response.send({ name: 'John Doe' })      // Content-type: application/json
```

## Sending a Response

The content-type can also be hardcoded

```
response.set('Content-Type', 'text/plain')  
response.send('Just regular text, no html expected!')
```

## Sending an Empty Response

```
response.status(404).end()
```

# Sessions

HTTP is a stateless protocol - information about a client is not retained over subsequent requests

Use sessions to overcome this problem

Enable the `cookieParser` and `session` middleware to process cookies

# Sessions

```
app.use(express.cookieParser())  
app.use(express.session({ secret: 'notastrongsecret' })))
```

The session is now accessible via `request.session`

```
app.get('...', function (request, response) {  
  var session = request.session  
})
```

## Redis Store with Express

```
$ npm install connect-redis express-session
```

```
var session = require('express-session'),  
    RedisStore = require('connect-redis')(session)
```

```
app.use(session({  
  store: new RedisStore(options),  
  secret: 'keyboard cat'  
}))
```

# Load-balancing

- Clusters
- Nginx
- HAProxy
- Varnish



# DEMO

RESTful API with Express: <https://github.com/azat-co/rest-api-express>



```
$ git clone https://github.com/azat-co/rest-api-express.git  
$ cd rest-api-express  
$ npm install  
$ node express.js
```

# Alternatives

- Sails
- LoopBack 🙌
- Meteor
- Hapi
- Restify

## More Alternatives

Registry of hand-picked Node frameworks: [nodeframework.com](http://nodeframework.com)

# Questions and Exercises



# Workshop



```
$ npm i -g expressworks
```

<https://github.com/azat-co/expressworks>

Videos for solutions: [YouTube ExpressWorks Playlist](#)

or <http://bit.ly/1jW1sBf>