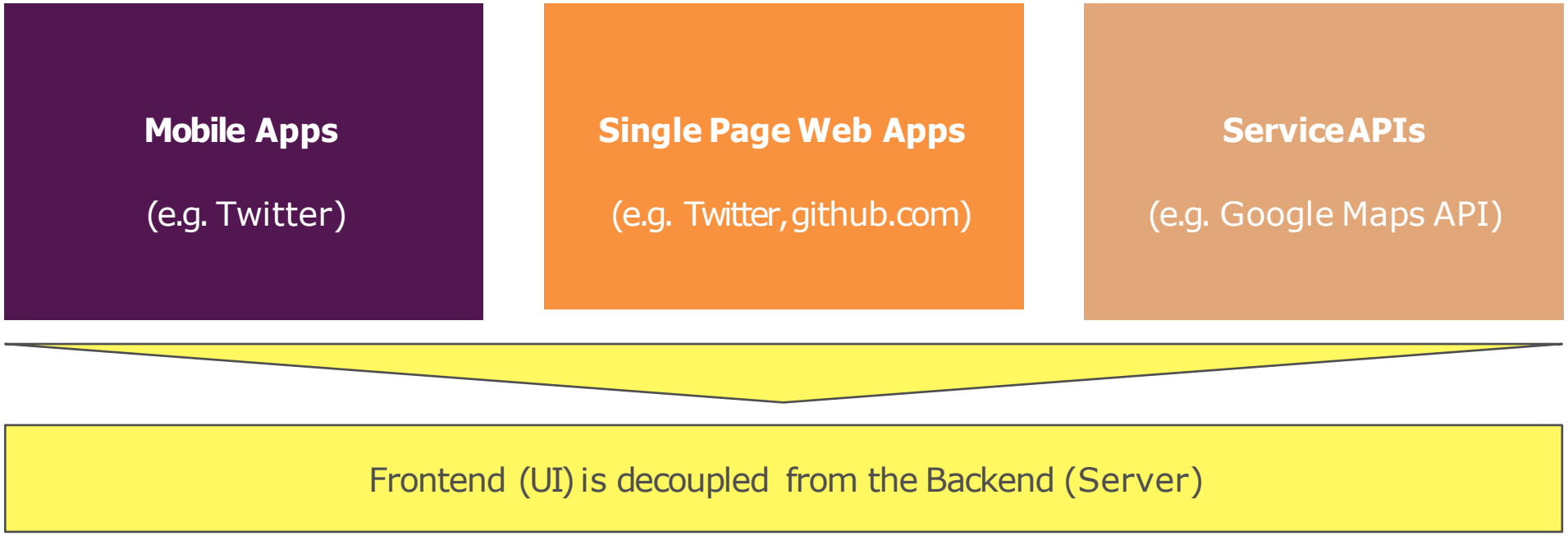
REST

What is REST?

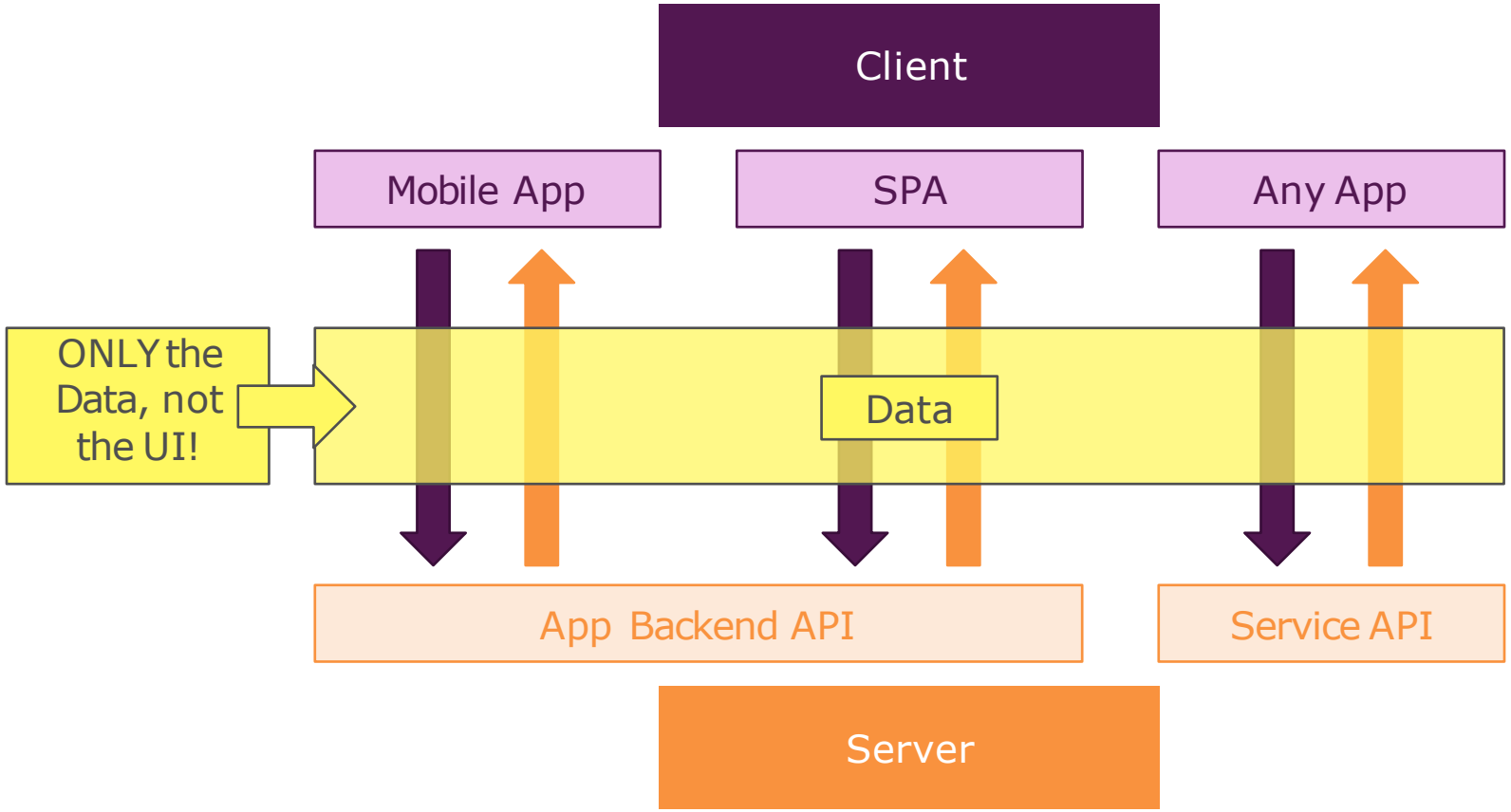
- REST = **RE**presentational **S**tate **T**ransfer
- REST is an architectural style consisting of a coordinated set of architectural constraints
- First described in 2000 by Roy Fielding in his doctoral dissertation at UC Irvine
- RESTful is typically used to refer to web services implementing a REST architecture
- Alternative to other distributed-computing specifications such as SOAP
- Simple HTTP client/server mechanism to exchange data
- Everything – the UNIVERSE is available through a URI
- Utilizes HTTP: GET/POST/PUT/DELETE operations

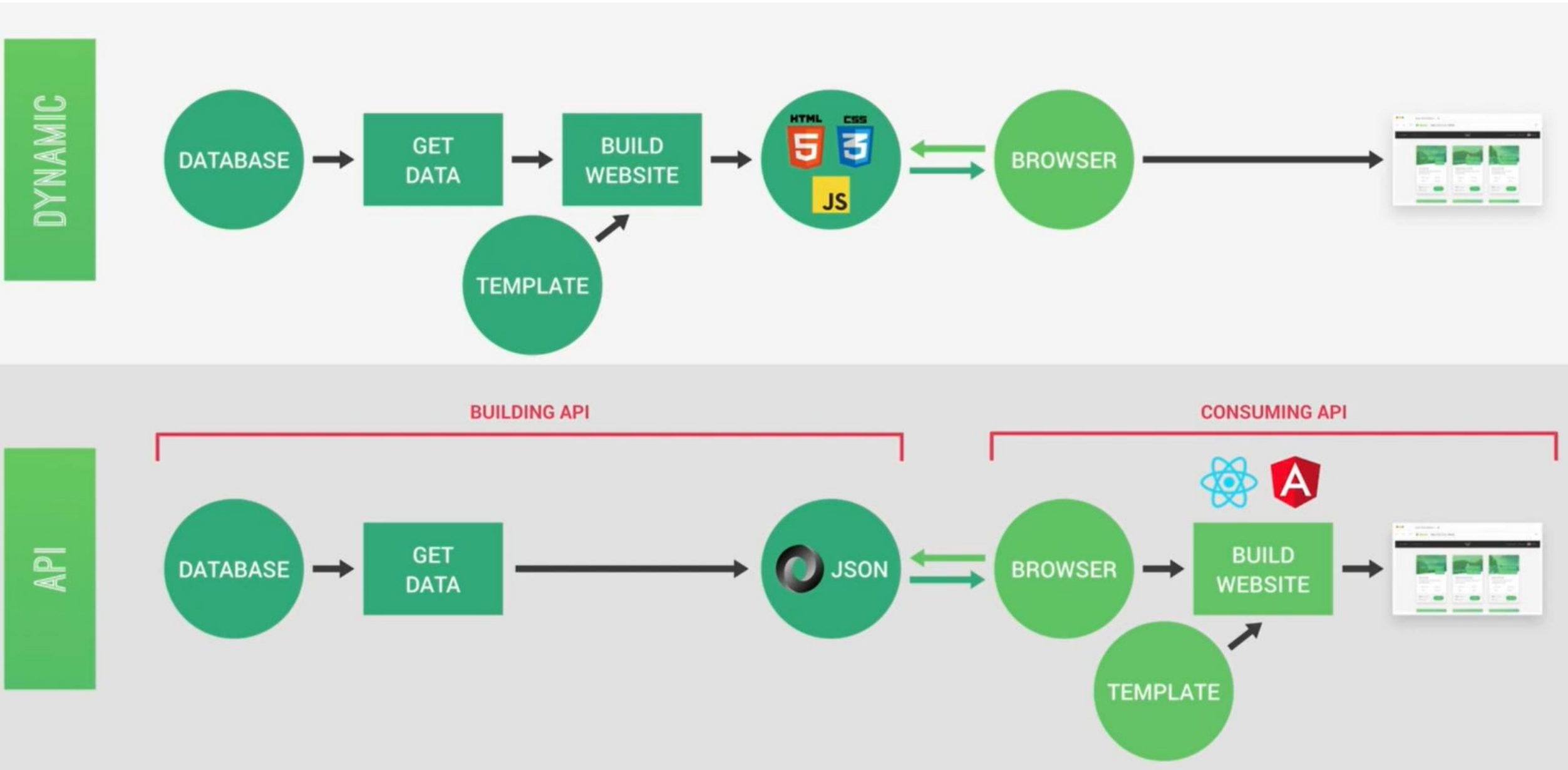
Why REST?

Not every Frontend (UI) requires HTMLPages!



REST API Big Picture




DYNAMIC
DATABASE
GET DATA
BUILD WEBSITE
TEMPLATE
HTML
CSS
JS
BROWSER
API
BUILDING API
CONSUMING API
DATABASE
GET DATA
JSON
BROWSER
BUILD WEBSITE
TEMPLATE

Data Formats

HTML	Plain Text	XML	JSON
`<p>Node.js</p>`	`Node.js`	`<name>Node.js</name>`	`{"title": "Node.js"}`
Data +Structure	Data	Data	Data
Contains User Interface	No UIAssumptions	No UIAssumptions	No UIAssumptions
Unnecessarily difficult to parse if you just need the data	Unnecessarily difficult to parse, no clear data structure	Machine-readable but relatively verbose; XML-parser needed	Machine-readable and concise; Can easily be converted to JavaScript

Architectural Constraints

- Client-server
  - Separation of concerns. A uniform interface separates clients from servers.
- Stateless
  - The client-server communication is further constrained by no client context being stored on the server between requests.
- Cacheable
  - Basic WWW principle: clients can cache responses.
- Uniform interface
  - Individual resources are identified in requests, i.e.,using URIs in web-based REST systems. REST APIs should have a uniform way of interacting with resources, regardless of the type of client.
- Layered system
  - A client cannot necessarily tell whether it is connected directly to the end server, or to an intermediary along the way.
- Code on demand (optional)
  - In some cases, servers can extend the functionality of clients by transferring executable code (e.g., JavaScript) to be executed on the client side.

Resource

- The key abstraction of information in REST is a **resource**.
  - a document or image, a temporal service, a collection of other resources, a non-virtual object (e.g. a person), and so on.

- Resource representation: consists of data, metadata describing the data and **hypermedia** links which can help the clients in transition to the next desired state.

https://restfulapi.net/rest-api-design-tutorial-with-example/

Resource Naming Best Practices -Use nouns to represent resources

- Document:
  - a **singular** concept, like an object instance or db record.
  - Use “singular” name to denote document resource archetype.
    - http://api.example.com/device-management/devices/{device-id}
    - http://api.example.com/user-management/users/{id}
    - http://api.example.com/user-management/users/admin
- Collection: sever-managed directory of resources.
  - Use “**plural**” name to denote collection resource archetype
    - http://api.example.com/device-management/devices
    - http://api.example.com/user-management/users
    - http://api.example.com/user-management/users/{id}/accounts

Resource Naming Best Practices -Consistency is the key

- **Use forward slash (/) to indicate hierarchical relationships**
  - The forward slash (/) character is used in the path portion of the URI to indicate a hierarchical relationship between resources.
  - http://api.example.com/device-management
  - http://api.example.com/device-management/devices
  - http://api.example.com/device-management/devices/{id}
- **Do not use trailing forward slash (/) in URIs**
  - http://api.example.com/device-management/devices/
  - http://api.example.com/device-management/devices /*This is much better version*/
- **Use hyphens (-) to improve the readability of URIs**
  - http://api.example.com/inventory-management/entities/{id}/script-locations //More readable
  - http://api.example.com/inventory-management/entities/{id}/scriptLocations //Less readable
  - http://api.example.com/inventory_management/entities/{id}/script_locations //Less readable
- **Use lowercase letters in URIs**

Resource Naming Best Practices -Never use CRUD function names in URIs

- HTTP request methods should be used to indicate which CRUD function is performed.
  - **Get all devices:**

    HTTP GET http://api.example.com/device-management/devices
  - **Create a new device:**

    HTTP POST http://api.example.com/device-management/devices
  - **Get a device for given id:**

    HTTP GET http://api.example.com/device-management/devices/{id}
  - **Update a device for given id:**

    HTTP PUT http://api.example.com/device-management/devices/{id}
  - **Delete a device for given id:**

    HTTP DELETE http://api.example.com/device-management/devices/{id}

Resource Naming Best Practices
-Use query component to filter URI collection

- Many times, you will come across requirements where you will need a collection of resources sorted, filtered or limited based on some certain resource attribute.

- For this, do not create new APIs – rather enable sorting, filtering and pagination capabilities in resource collection API and pass the input parameters as query parameters. e.g.

  - http://api.example.com/device-management/devices
  - http://api.example.com/device-management/devices?region=USA
  - http://api.example.com/device-management/devices?region=USA&brand=XYZ

Route

A route consists of the following parts:

- HTTP verb
- Entity
- Params
- ~~Query Params~~

Example:

```
GET /users/:user_id
```

will match any request to the following URL:

```
GET /users/1
GET /users/2?name=anna
```

The the following requests will not match:

```
GET /users/
GET /users/?name=anna
```

HTTP Verbs and CRUD Consistency

The following are the most commonly used server architecture HTTP methods and their corresponding Express methods:

GET app.get() Retrieves an entity or a list of entities
POST app.post() Submits a new entity
PUT app.put() Updates an entity by complete replacement
DELETE app.delete() Delete an existing entity

HTTP Methods for RESTful APIs

HTTP METHOD	CRUD	ENTIRE COLLECTION (E.G. /USERS)	SPECIFIC ITEM (E.G. /USERS/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID.	Avoid using POST on single resource
GET	Read	200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single user. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resource.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution.	200 (OK). 404 (Not Found), if ID not found or invalid.

http status code: https://restfulapi.net/http-status-codes/

idempotent and safe HTTP methods

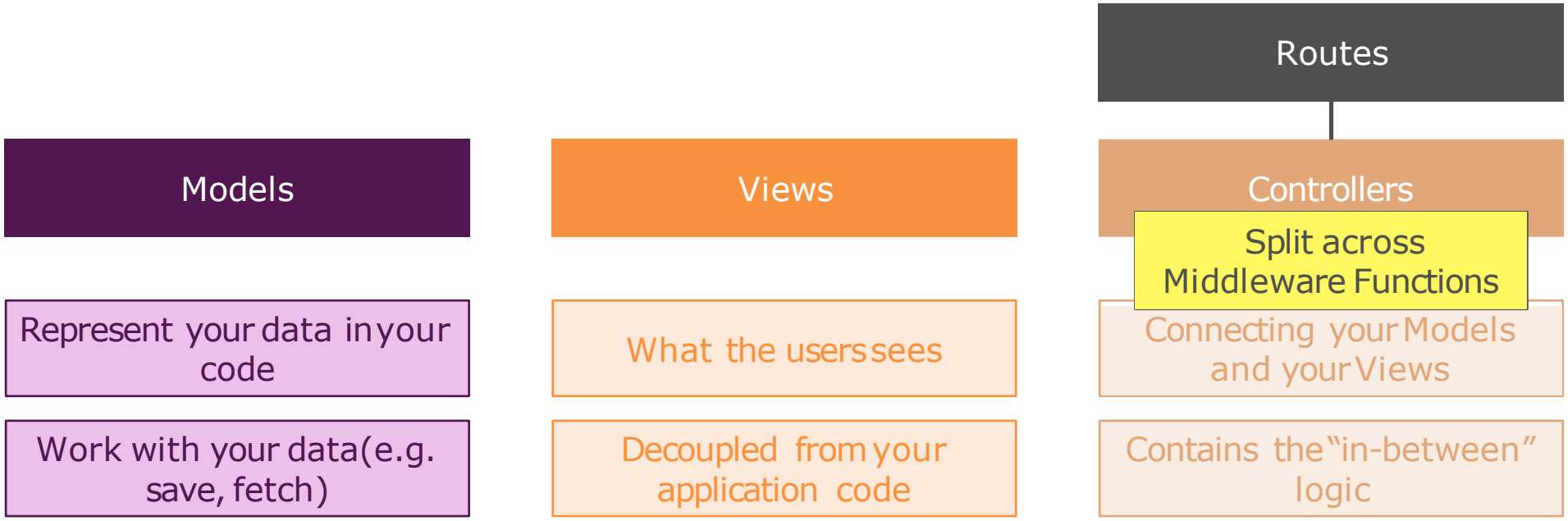
- idempotent: if making identical requests have the same effect as making a single request.
- idempotent methods will not throw different outcomes even if you call them multiple times.
  - They will always return the same result unless you change the URL.

- safe methods don't change the representation of the resource in the Server.

Method	Safe	Idempotent
GET	Yes	Yes
PUT	No	Yes
POST	No	No
DELETE	No	Yes

What's MVC?

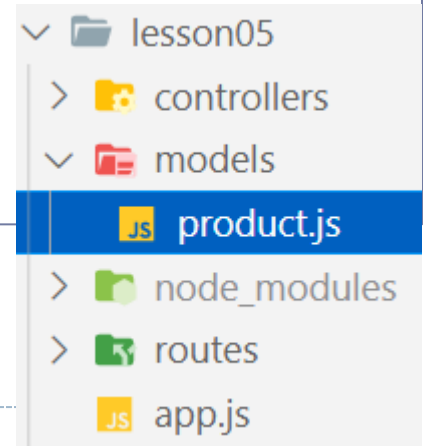
Separation of Concerns



Demo: Shopping Cart - Model

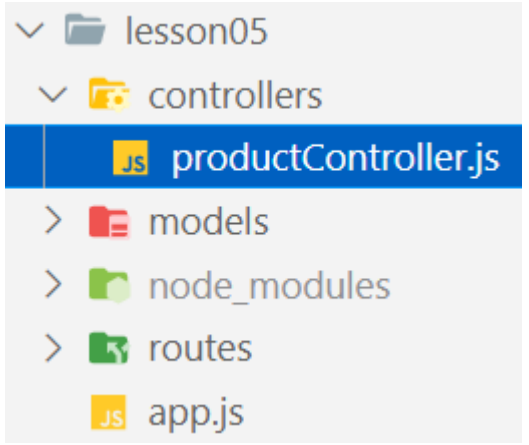
```
let products = [];

export class Product {

    constructor(id, title, price, description) {
        this.id = id;
        this.title = title;
        this.price = price;
        this.description = description;
    }

    save() {
        this.id = Math.random().toString();
        products.push(this);
        return this;
    }

    update() {
        const index = products.findIndex(p => p.id === this.id);
        if (index > -1) {
            products.splice(index, 1, this);
            return this;
        } else {
            throw new Error('NOT Found');
        }

    }
```

```
    static fetchAll() {
            return products;
        }

        static findById(productId) {
            const index = products.findIndex(p => p.id === productId);
            if (index > -1) {
                return products[index];
            } else {
                throw new Error('NOT Found');
            }
        }

        static deleteById(productId) {
            const index = products.findIndex(p => p.id === productId);
            if (index > -1) {
                products = products.filter(p => p.id !== productId);
            } else {
                throw new Error('NOT Found');
            }
        }

}
```



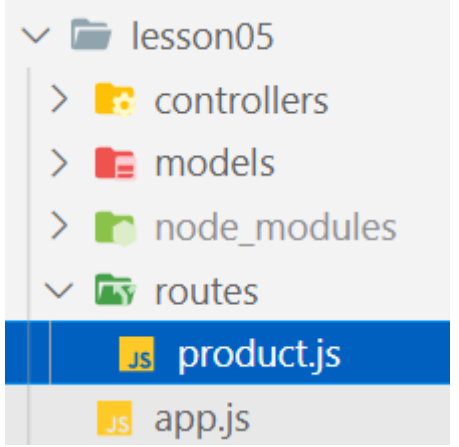
Demo: Shopping Cart – Controller

```
import Product from './models/product';

export getProducts = (req, res, next) => {
    res.status(200).json(Product.fetchAll());
}

export getProductById = (req, res, next) => {
    res.status(200).json(Product.findById(req.params.prodId));
}

export save = (req, res, next) => {
    const prod = req.body;
    const savedProd = new Product(null, prod.title, prod.price, prod.description).save();
    res.status(201).json(savedProd);
}

export update = (req, res, next) => {
    const prod = req.body;
    const updatedProd = new Product(req.params.prodId, prod.title, prod.price, prod.description).update();
    res.status(200).json(updatedProd);
}

export deleteById = (req, res, next) => {
    Product.deleteById(req.params.prodId);
    res.status(200).end();
}
```



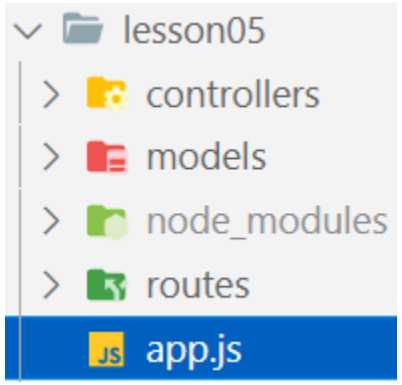
Demo: Shopping Cart – Route

```
import express from 'express’;
import productController from './controllers/productController';

const router = express.Router();

router.get('/', productController.getProducts);

router.get('/:prodId', productController.getProductById);

router.post('/', productController.save);

router.put('/:prodId', productController.update);

router.delete('/:prodId', productController.deleteById);

export router;
```

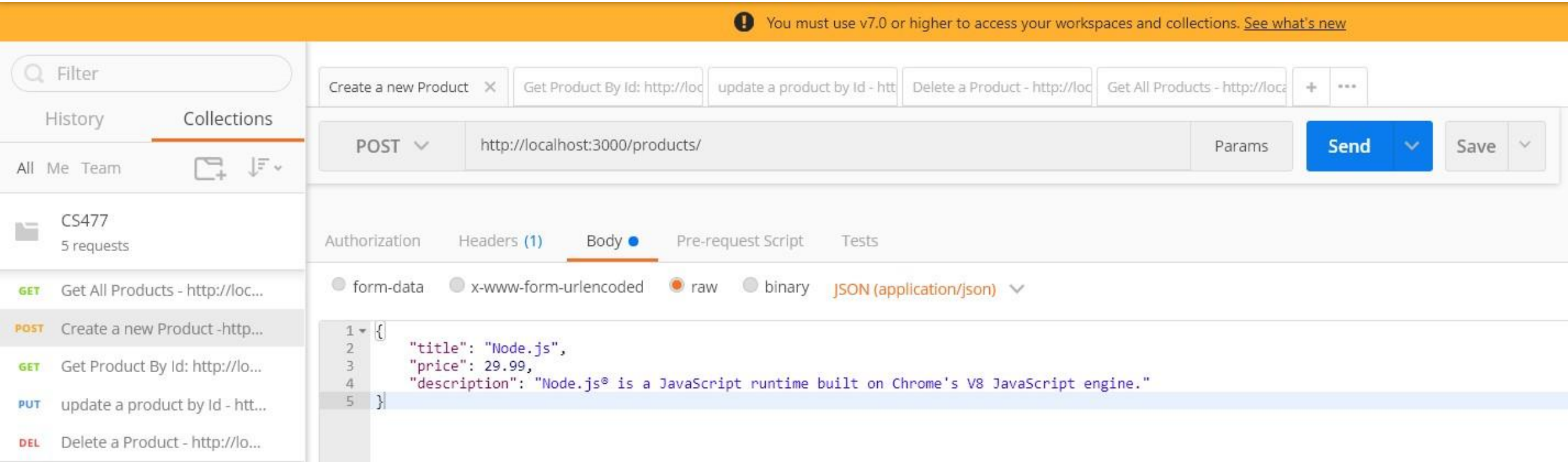


Demo: Shopping Cart – app.js

```
import express from 'express';
import productRouter from './routes/product';
const cors = require('cors');

const app = express();

app.use(cors());
app.use(express.json());

app.use('/products', productRouter);

app.use((req, res, next) => {
    res.status(404).json({ error: req.url + ' API not supported!' });
});

app.use((err, req, res, next) => {
    if (err.message === 'NOT Found') {
        res.status(404).json({ error: err.message });
    } else {
        res.status(500).json({ error: 'Something is wrong! Try later' });
    }
});

app.listen(3000, () => console.log('listening to 3000...'));
```

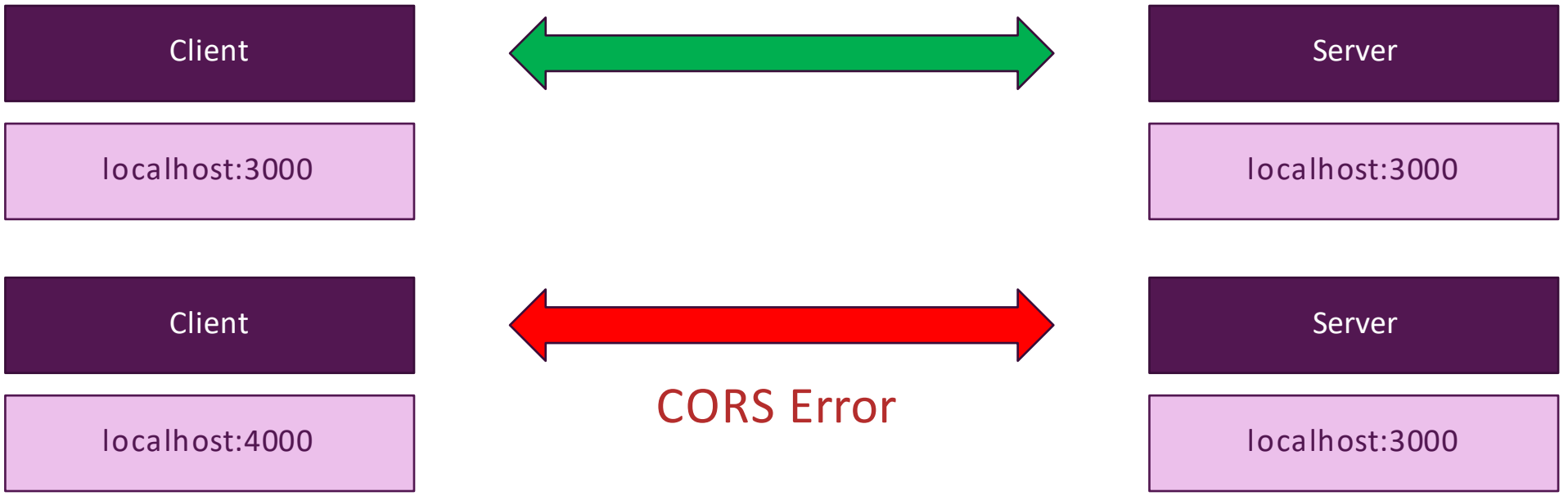


Demo: Shopping Cart – Testing APIs



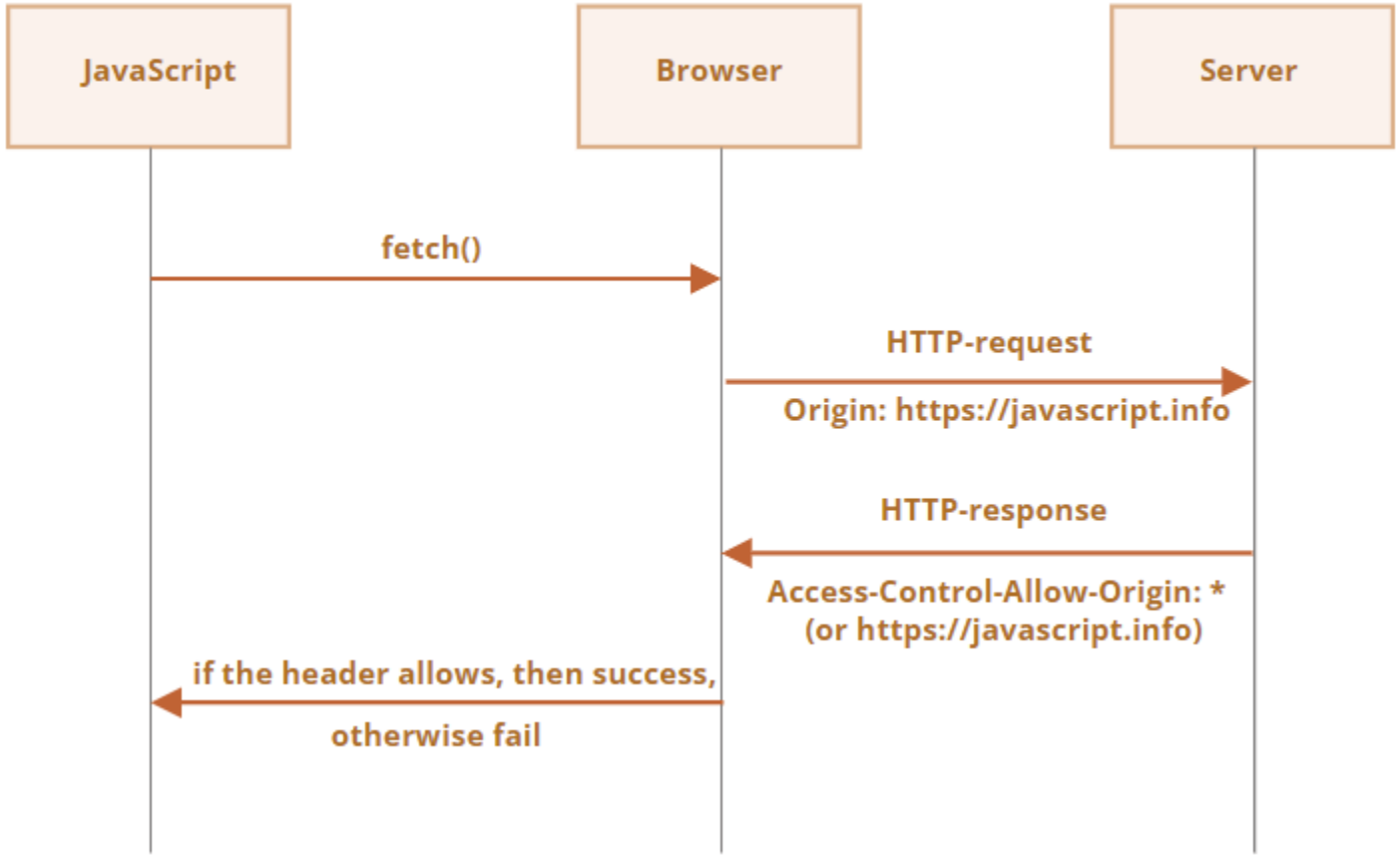
CORS (Cross-Origin Resource Sharing)

- **Cross-Origin Resource Sharing** (CORS) is a mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own.



cross-origin request using Fetch API

- **The browser plays the role of a trusted mediator here:**



cors

- npm install cors

- **Simple Usage (Enable *All* CORS Requests)**
  - `const cors = require('cors');`
  - `app.use(cors());`

- **Enable CORS for a Single Route**
  - `router.post('/users', cors(),userController.insert);`