Sprint 1
Dynamic Price Calculations

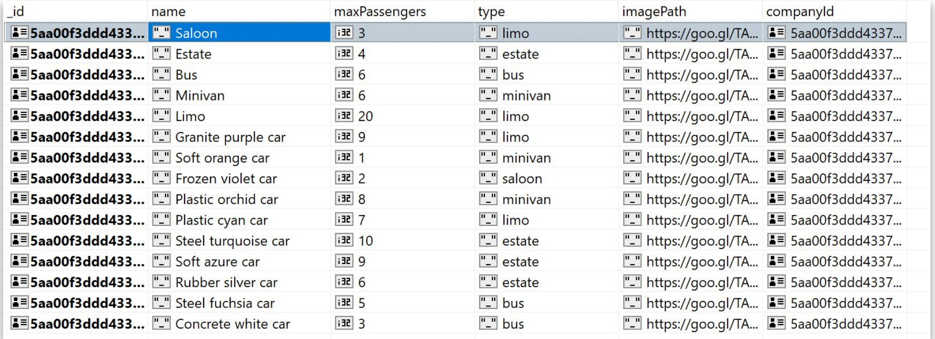
Most complex but basic calculation using companyId, vehicleTypes, maxPassengers, enabled pricing rules ordered by precedence, trip distance and duration

Data Fixtures

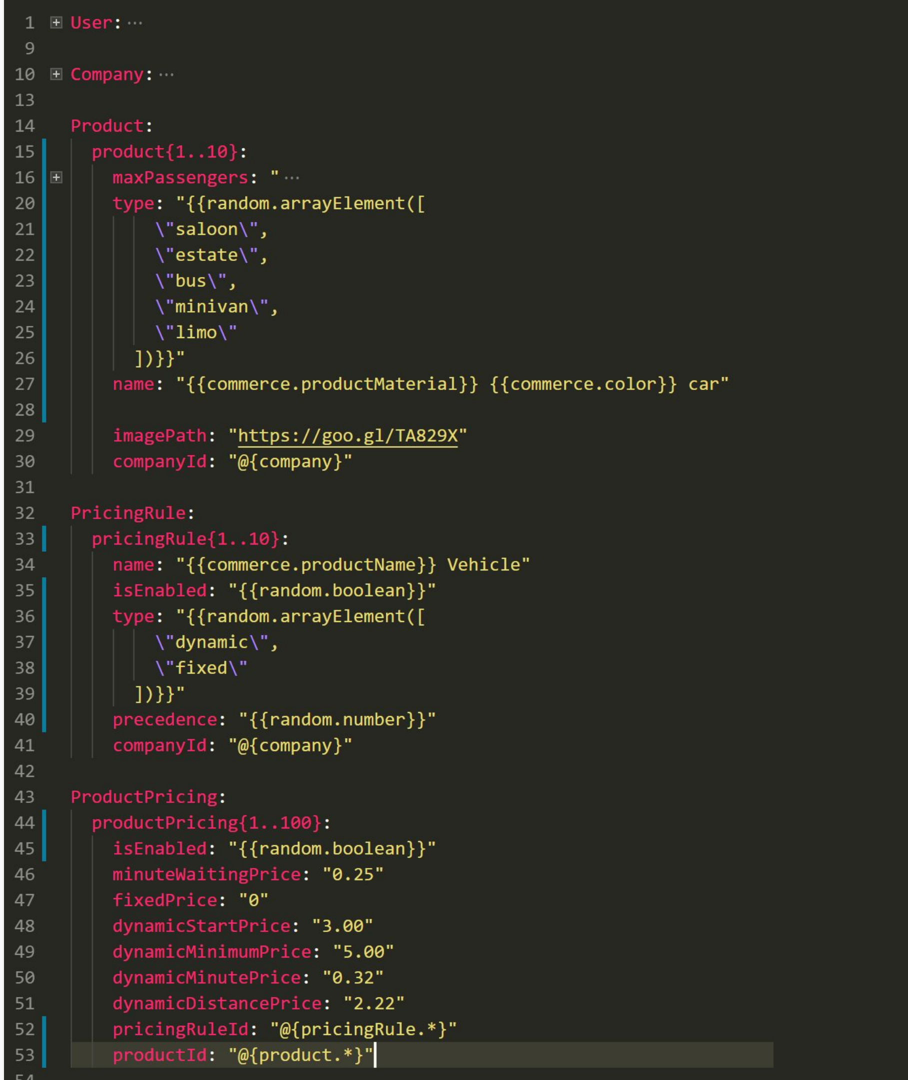
Fake data is generated so that the process of developing the dynamic price calculation is consistent and swift across developer machines

Faking Data

Random data is inserted in the database using data fixtures



_id	name	maxPassengers	type	imagePath	companyId
5aa00f3ddd433...	Saloon	3	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Estate	4	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Bus	6	bus	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Minivan	6	minivan	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Limo	20	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Granite purple car	9	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Soft orange car	1	minivan	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Frozen violet car	2	saloon	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Plastic orchid car	8	minivan	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Plastic cyan car	7	limo	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Steel turquoise car	10	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Soft azure car	9	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Rubber silver car	6	estate	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Steel fuchsia car	5	bus	https://goo.gl/TA...	5aa00f3ddd4337...
5aa00f3ddd433...	Concrete white car	3	bus	https://goo.gl/TA...	5aa00f3ddd4337...



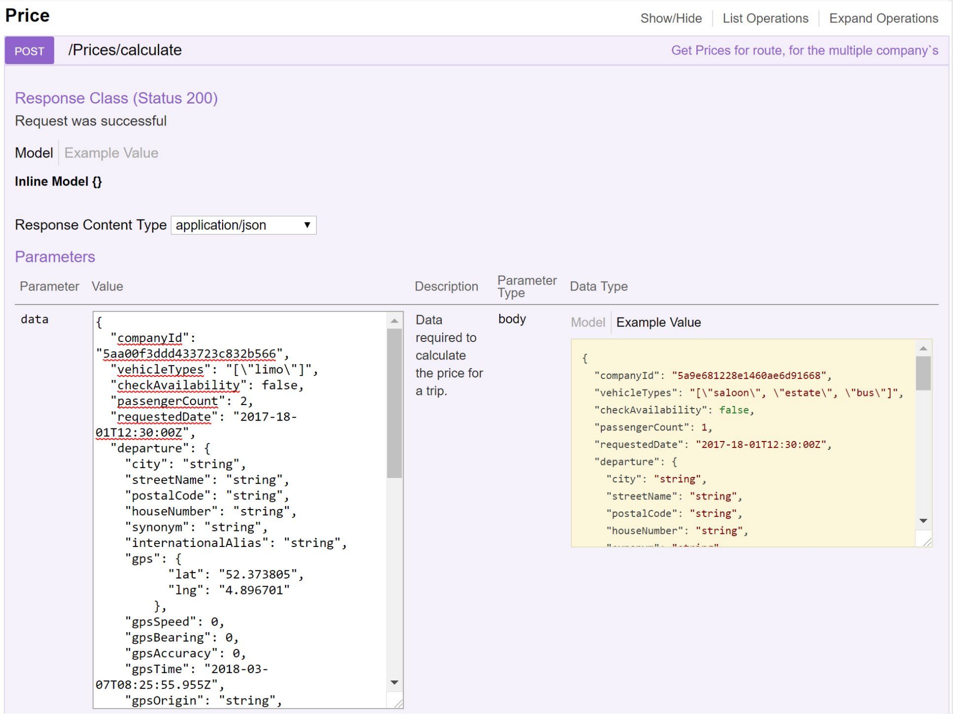
```
User: ···

Company: ···

Product:
  product{1..10}:
    maxPassengers: "···
    type: "{{random.arrayElement([
        \"saloon\",
        \"estate\",
        \"bus\",
        \"minivan\",
        \"limo\"
      ])}}"
    name: "{{commerce.productMaterial}} {{commerce.color}} car"

    imagePath: "https://goo.gl/TA829X"
    companyId: "@{company}"

PricingRule:
  pricingRule{1..10}:
    name: "{{commerce.productName}} Vehicle"
    isEnabled: "{{random.boolean}}"
    type: "{{random.arrayElement([
        \"dynamic\",
        \"fixed\"
      ])}}"
    precedence: "{{random.number}}"
    companyId: "@{company}"

ProductPricing:
  productPricing{1..100}:
    isEnabled: "{{random.boolean}}"
    minuteWaitingPrice: "0.25"
    fixedPrice: "0"
    dynamicStartPrice: "3.00"
    dynamicMinimumPrice: "5.00"
    dynamicMinutePrice: "0.32"
    dynamicDistancePrice: "2.22"
    pricingRuleId: "@{pricingRule.*}"
    productId: "@{product.*}"
```

Price Calculation

In this first sprint, the basic steps of a dynamic price calculation are orchestrated

Step 1 - PassengerApp sends request to TPS

The next couple of slides show the process of sending the request to our TPS service, and the way that our server processes the request before returning a response with a price calculation for each requested product



Price

/Prices/calculate

Get Prices for route, for the multiple company`s

Response Class (Status 200)

Request was successful

Inline Model {}

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
data		Data required to calculate the price for a trip.	body	Model / Example Value

```
{
  "companyId":
"5aa00f3ddd433723c832b566",
  "vehicleTypes": "[\"limo\"]",
  "checkAvailability": false,
  "passengerCount": 2,
  "requestedDate": "2017-18-
01T12:30:00Z",
  "departure": {
    "city": "string",
    "streetName": "string",
    "postalCode": "string",
    "houseNumber": "string",
    "synonym": "string",
    "internationalAlias": "string",
    "gps": {
          "lat": "52.373805",
          "lng": "4.896701"
        },
    "gpsSpeed": 0,
    "gpsBearing": 0,
    "gpsAccuracy": 0,
    "gpsTime": "2018-03-
07T08:25:55.955Z",
    "gpsOrigin": "string",
```

```
{
  "companyId": "5a9e681228e1460ae6d91668",
  "vehicleTypes": "[\"saloon\", \"estate\", \"bus\"]",
  "checkAvailability": false,
  "passengerCount": 1,
  "requestedDate": "2017-18-01T12:30:00Z",
  "departure": {
    "city": "string",
    "streetName": "string",
    "postalCode": "string",
    "houseNumber": "string",
    "synonym": "string"
```

Request

As the documentation of the old system suggests, the query format in the yellow box is expected, and used as an example

Source: https://docs.dispatchapi.io/#get-prices-per-vehicle-type

```
{
    companyId:
    vehicleTypes:
    passengerCount:
    departure: {
        gps: {
            lat:
            lng:
        }
    destination: {
        gps: {
            lat:
            lng:
        }
    }
}
```

Data

The values on the left side of this slide are the only values that are currently being accepted by the endpoint

Step 2 - Obtaining ride distance and duration

The distance and duration of a trip are provided by the google directions API

The next slide shows the request parameters sent to google directions API, and the desired response attributes

Request

Fields used in google directions:

1. departure (gps: lat, lng)
2. destination (gps: lat, lng)

Response

Returned by google:

1. distance (in m)
2. duration (in s)

Step 3 - Querying our database for matches

While location matching is not part of the system yet, we could theoretically pass all the information we have at this moment to our database query to get the best possible match while ignoring the locations and timeframes for now

The query is performed for every vehicle type that the user wants to see, and returns exactly one best result for each

The next slide shows the request that would be sent by the Passenger App to our TPS microservice

Query

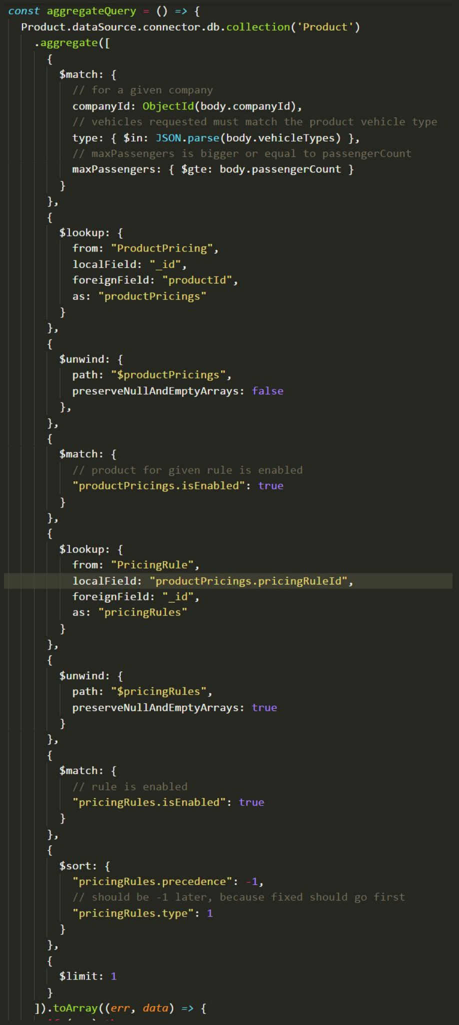
Fields used in query:

1. companyId
2. vehicleTypes
3. passengerCount

Fields unused:

4. departure
5. destination
6. pickupTime

This query grows when complexity of the application increases



```
const aggregateQuery = () => {
  Product.dataSource.connector.db.collection('Product')
    .aggregate([
      {
        $match: {
          // for a given company
          companyId: ObjectId(body.companyId),
          // vehicles requested must match the product vehicle type
          type: { $in: JSON.parse(body.vehicleTypes) },
          // maxPassengers is bigger or equal to passengerCount
          maxPassengers: { $gte: body.passengerCount }
        }
      },
      {
        $lookup: {
          from: "ProductPricing",
          localField: "_id",
          foreignField: "productId",
          as: "productPricings"
        }
      },
      {
        $unwind: {
          path: "$productPricings",
          preserveNullAndEmptyArrays: false
        },
      },
      {
        $match: {
          // product for given rule is enabled
          "productPricings.isEnabled": true
        }
      },
      {
        $lookup: {
          from: "PricingRule",
          localField: "productPricings.pricingRuleId",
          foreignField: "_id",
          as: "pricingRules"
        }
      },
      {
        $unwind: {
          path: "$pricingRules",
          preserveNullAndEmptyArrays: true
        }
      },
      {
        $match: {
          // rule is enabled
          "pricingRules.isEnabled": true
        }
      },
      {
        $sort: {
          "pricingRules.precedence": -1,
          // should be -1 later, because fixed should go first
          "pricingRules.type": 1
        }
      },
      {
        $limit: 1
      }
    ]).toArray((err, data) => {
```

Step 4 - Calculating the prices

After the query to the database has been made, the most complex work is done to calculate prices based on different rules provided and stored in our database by the group admins

A group admin can choose whether he would like the price to be calculated using tiers. He can flip a switch after he’s defined the thresholds and tier prices for every one of his products

E.g. $0.5 dollar per km for the first 10 km, plus $0.4 * the next 10 km, plus $0.35 for the rest 2.54 km.
total = 5 + 4 + 0.889
final = max(9.89 + 3, 5)
final = 9.89

(this example only uses the distance metric)

```
total =

    metric * metricPrice

    or if tier pricing

    each(threshold * thresholdPrice)

final = max(
    total + startAmount,
    minAmount
)
```

total: km * kmPrice

or

km - thresholds * kmPrice
+ (threshold * tierPrice)

Final: the price that is finally returned

Step 5 - Sending back the response

When all the prices have been calculated (for each vehicle type / product), the response is sent back to the PassengerApp

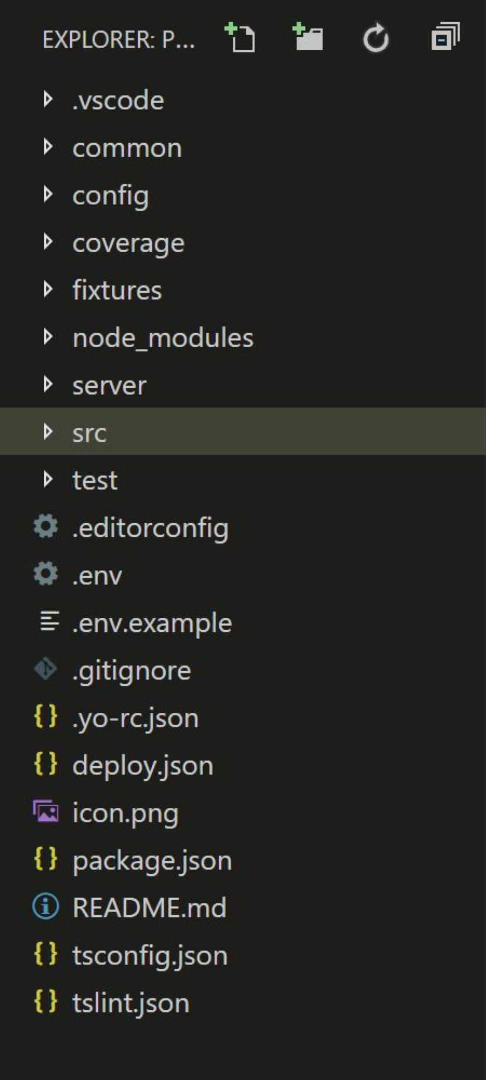
Response

Each vehicle type / product has a maximum of one result

```
[
  {
    "vehicleType": "saloon",
    "maxPassengers": 8,
    "fixedPrice": true,
    "price": {
      "currency": "EUR",
      "total": 1165,
      "breakdown": {
        "route": 1099,
        "tax": 66,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": 0
      }
    }
  },
  {
    "vehicleType": "limo",
    "maxPassengers": 5,
    "fixedPrice": true,
    "price": {
      "currency": "EUR",
      "total": 1165,
      "breakdown": {
        "route": 1099,
        "tax": 66,
        "toll": 0,
        "parking": 0,
        "waiting": 0,
        "discount": 0
      }
    }
  },
```

Project Structure

As Loopback 3 does not support Typescript out of the box, a separation between inherent loopback files and external functionality files is made, so that Typescript can be used for pieces of software that are decoupled from the framework



- .vscode
- common
- config
- coverage
- fixtures
- node_modules
- server
- src
- test
- .editorconfig
- .env
- .env.example
- .gitignore
- .yo-rc.json
- deploy.json
- icon.png
- package.json
- README.md
- tsconfig.json
- tslint.json

File Structure

common	Loopback models & schemas
config	Loopback config files
coverage	Test reporting
fixtures	Data fixtures for generating test data in db
server	Loopback server files
src	Typescript project
test	Typescript tests
.editorconfig .env .tsconfig .tslint	Space, tabs, line-ending styles Environmental variables Typescript settings Typescript linting

Tests

Tests are written using Mocha and Chai to guarantee that a functionalities continue to operate consistently, adhering to the FIRST mnemonic

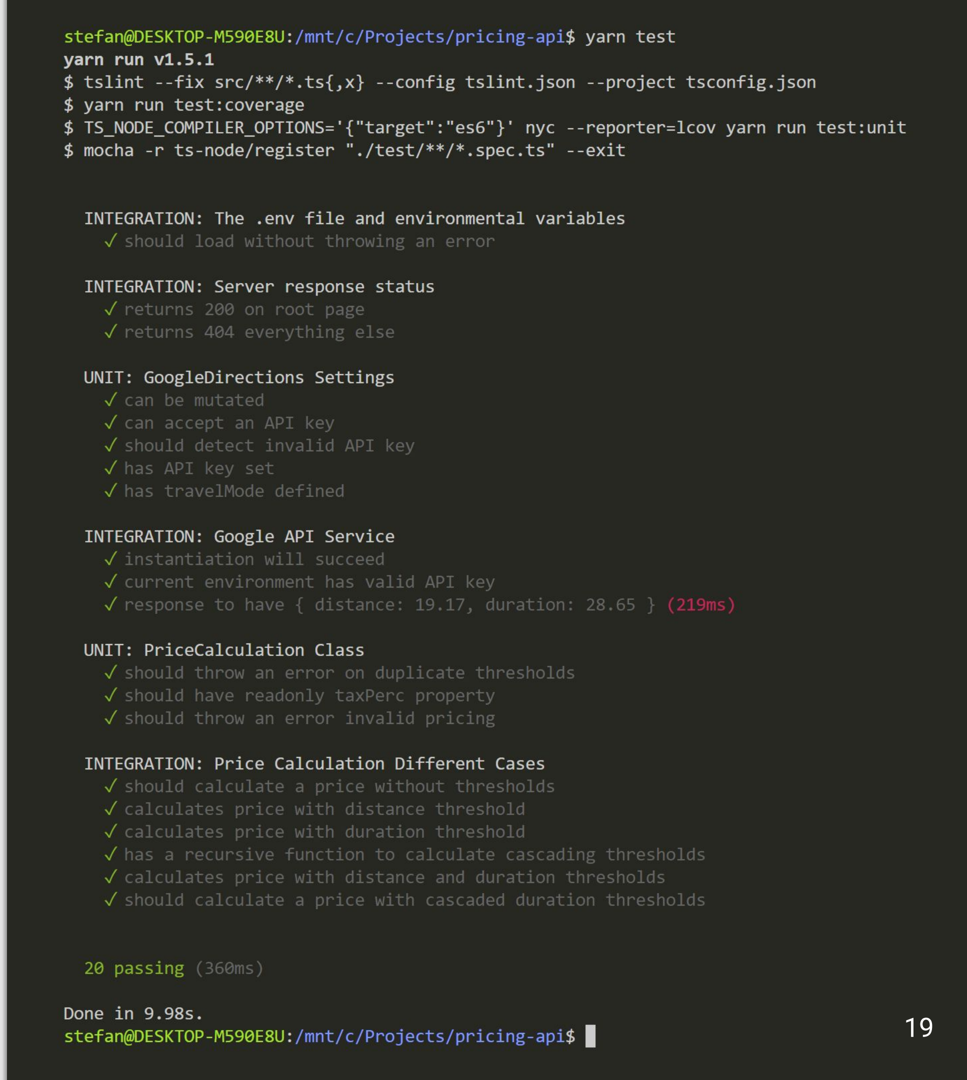
1. Fast
2. Isolate
3. Repeatable
4. Self validating
5. Timely

Output

1. UNIT:
   Aims to test small units of code

2. INTEGRATION:
   Tests whether different parts of the system work together

3. Note:
   Current tests assume that the environment in which it resides is operational. For example: a google directions api key is set, the system is connected to the network, et cetera.

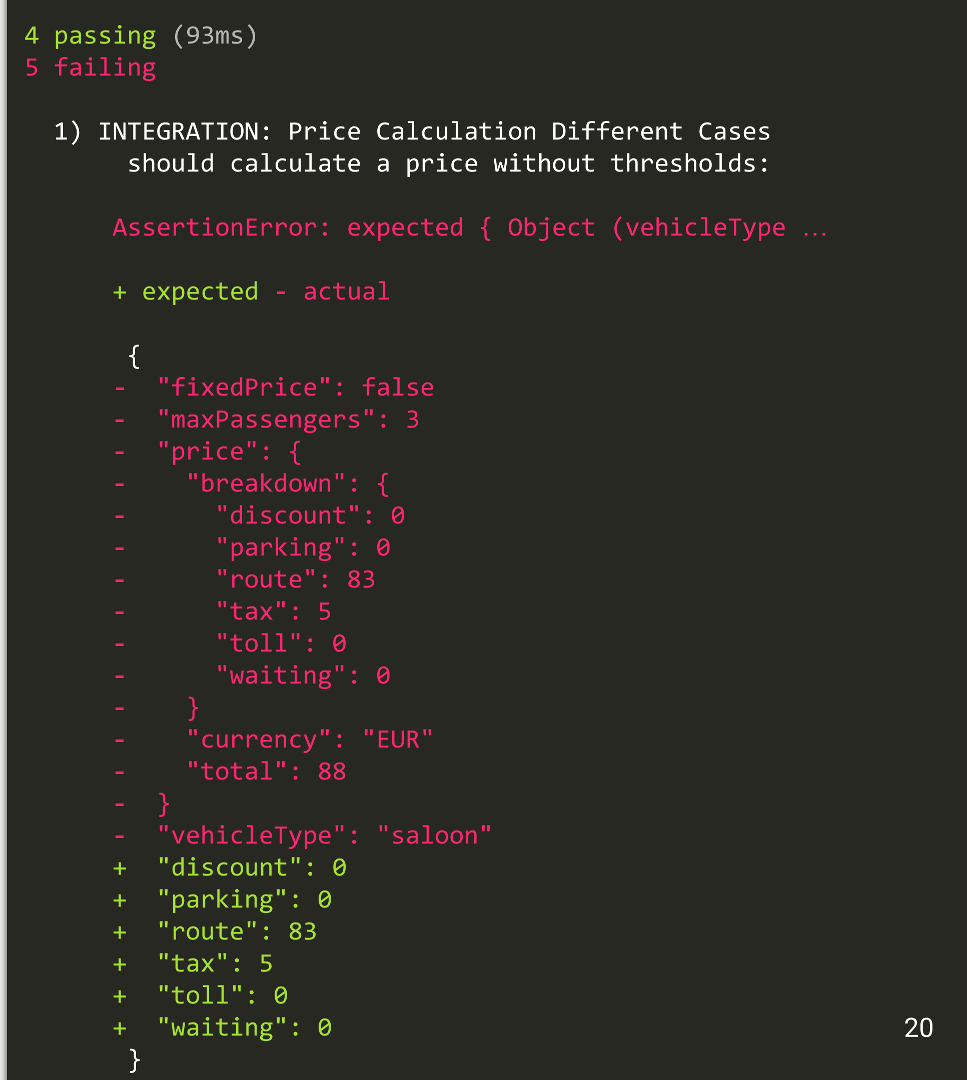


```
stefan@DESKTOP-M590E8U:/mnt/c/Projects/pricing-api$ yarn test
yarn run v1.5.1
$ tslint --fix src/**/*.ts{,x} --config tslint.json --project tsconfig.json
$ yarn run test:coverage
$ TS_NODE_COMPILER_OPTIONS='{"target":"es6"}' nyc --reporter=lcov yarn run test:unit
$ mocha -r ts-node/register "./test/**/*.spec.ts" --exit


  INTEGRATION: The .env file and environmental variables
    ✓ should load without throwing an error

  INTEGRATION: Server response status
    ✓ returns 200 on root page
    ✓ returns 404 everything else

  UNIT: GoogleDirections Settings
    ✓ can be mutated
    ✓ can accept an API key
    ✓ should detect invalid API key
    ✓ has API key set
    ✓ has travelMode defined

  INTEGRATION: Google API Service
    ✓ instantiation will succeed
    ✓ current environment has valid API key
    ✓ response to have { distance: 19.17, duration: 28.65 } (219ms)

  UNIT: PriceCalculation Class
    ✓ should throw an error on duplicate thresholds
    ✓ should have readonly taxPerc property
    ✓ should throw an error invalid pricing

  INTEGRATION: Price Calculation Different Cases
    ✓ should calculate a price without thresholds
    ✓ calculates price with distance threshold
    ✓ calculates price with duration threshold
    ✓ has a recursive function to calculate cascading thresholds
    ✓ calculates price with distance and duration thresholds
    ✓ should calculate a price with cascaded duration thresholds


  20 passing (360ms)

Done in 9.98s.
stefan@DESKTOP-M590E8U:/mnt/c/Projects/pricing-api$ █
```

Debugging

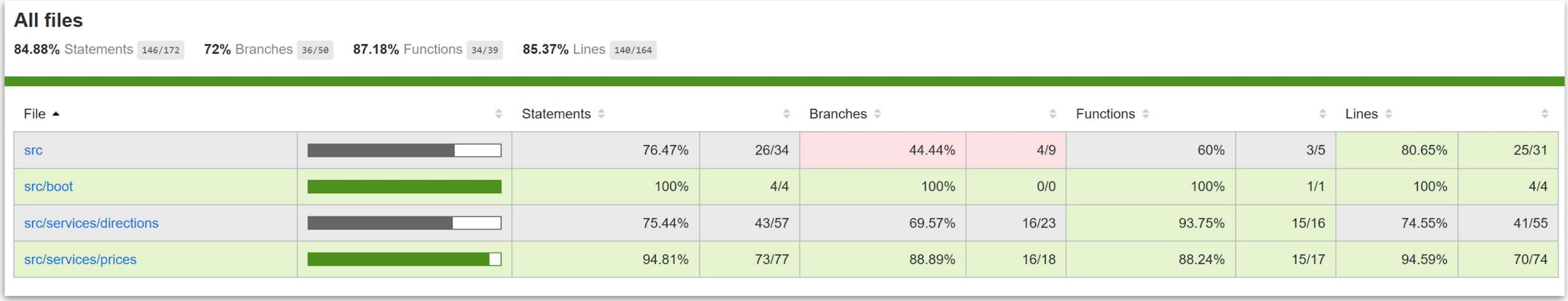
Set the debug flag to true to display errors and logs during the tests

```
import debug from '../../debug';

debug(true);

describe('UNIT: PriceCalculation Class', () => {

 it('should throw an error on duplicate...
```



```
4 passing (93ms)
5 failing

  1) INTEGRATION: Price Calculation Different Cases
       should calculate a price without thresholds:

      AssertionError: expected { Object (vehicleType ...

      + expected - actual

       {
      -   "fixedPrice": false
      -   "maxPassengers": 3
      -   "price": {
      -     "breakdown": {
      -       "discount": 0
      -       "parking": 0
      -       "route": 83
      -       "tax": 5
      -       "toll": 0
      -       "waiting": 0
      -     }
      -     "currency": "EUR"
      -     "total": 88
      -   }
      -   "vehicleType": "saloon"
      +   "discount": 0
      +   "parking": 0
      +   "route": 83
      +   "tax": 5
      +   "toll": 0
      +   "waiting": 0
       }
```

Tests: coverage reporting

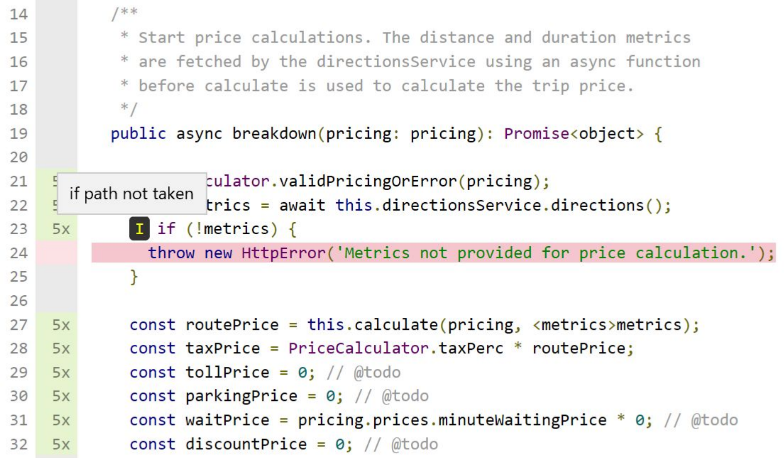


All files

84.88% Statements 146/172 **72%** Branches 36/50 **87.18%** Functions 34/39 **85.37%** Lines 140/164

File		Statements		Branches		Functions		Lines	
src		76.47%	26/34	44.44%	4/9	60%	3/5	80.65%	25/31
src/boot		100%	4/4	100%	0/0	100%	1/1	100%	4/4
src/services/directions		75.44%	43/57	69.57%	16/23	93.75%	15/16	74.55%	41/55
src/services/prices		94.81%	73/77	88.89%	16/18	88.24%	15/17	94.59%	70/74

Istanbul tests checks to see what lines of code were run. The report shows useful information to improve the test coverage of a project.



```
        /**
         * Start price calculations. The distance and duration metrics
         * are fetched by the directionsService using an async function
         * before calculate is used to calculate the trip price.
         */
        public async breakdown(pricing: pricing): Promise<object> {

5                       culator.validPricingOrError(pricing);
5                       trics = await this.directionsService.directions();
5x        I if (!metrics) {
            throw new HttpError('Metrics not provided for price calculation.');
          }

5x        const routePrice = this.calculate(pricing, <metrics>metrics);
5x        const taxPrice = PriceCalculator.taxPerc * routePrice;
5x        const tollPrice = 0; // @todo
5x        const parkingPrice = 0; // @todo
5x        const waitPrice = pricing.prices.minuteWaitingPrice * 0; // @todo
5x        const discountPrice = 0; // @todo
```

if path not taken