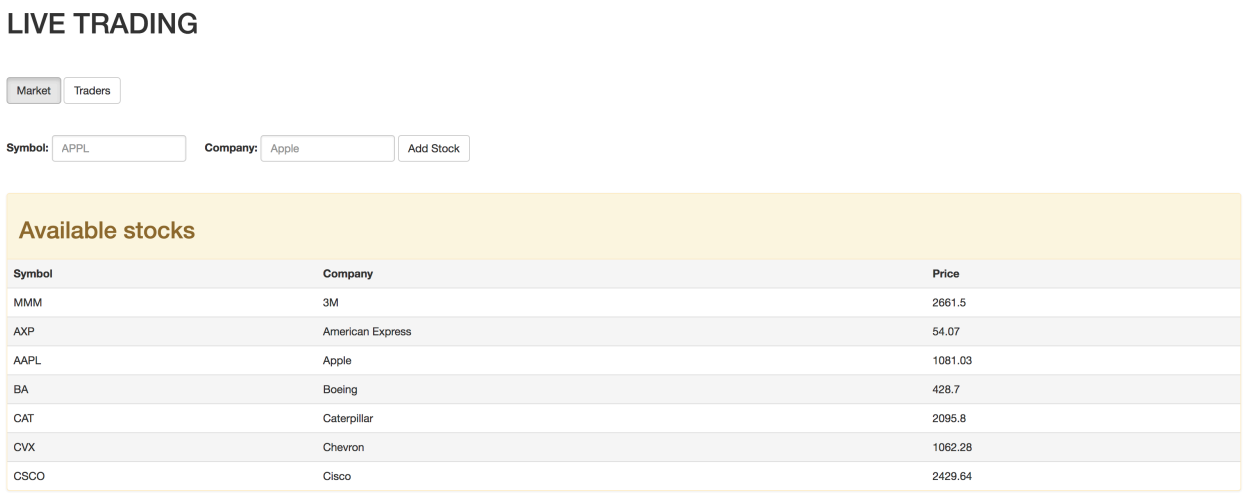
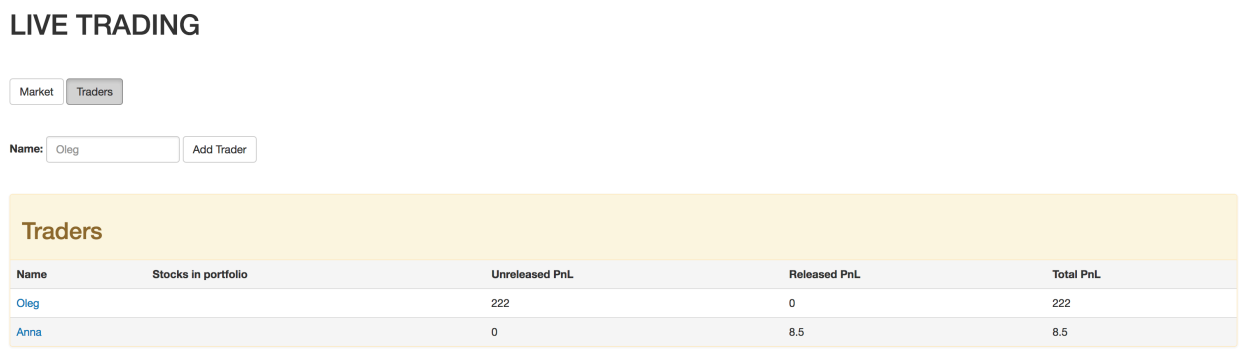
Introduction

Live Trading Application

1. Market screen, where you can see the list of stocks with live prices and add new stock to the market.



2. Traders screen where you can add new traders to the system and look for trader statistics.



You can click to “Market” or “Traders” button to navigate between these views.

3. When you click on trader name inside traders table you will navigate to Trader Details view. Where you can find the lists of Open and Closed positions, current Profit and Loss statistics, make new Trade or Close open position.

LIVE TRADING

Market | Traders

Oleg Back

Unreleased PnL: 505.3

Released PnL: 0

Total PnL: 505.3

Symbol

Count

10

Buy St

MMM
AXP
AAPL
BA
CAT

Open positions

#	Symbol	Mark	Count	Unreleased PnL	
0	MMM 3M	1097.06	10	276.2	Close
1	AAPL Apple	1062.59	10	229.1	Close

Closed Trades

#	Symbol	Mark	Close Price	Count	PnL

Step 1. Project Initialization

1. Download and Install latest version of **Node.js** from https://nodejs.org/en/
2. Run **npm install -g @angular/cli**
3. Run **ng new trading-app**

New project structure should be created for now.

4. Run **Intellij IDEA** and open generated project.

5. Open **Terminal** window via IDEA

6. Run **npm run start**

7. Open **http://localhost:4200/**

Step 2. Market Viewer

Go to **app** folder and create new directory **domain**, we will keep all the data objects here.

Create new TypeScript file with name **Stock**.

```
export class Stock
{
  constructor(private symbol: string, private company: string) { }

  getSymbol(): string
  {
    return this.symbol;
  }

  getCompany(): string
  {
    return this.company;
  }

  getPrice(): number
  {
    return 0;
  }
}
```

Open **AppComponent** class add **stocks** array and push some mock stocks inside using **ngOnInit()** method implemented from Angular **OnInit** interface. This method will be executed right after component creation.

Note: **Stock** class should be imported from **domain** folder to became available here.

```
import {Stock} from "./domain/Stock";

export class AppComponent implements OnInit
{
  stocks: Stock[];

  ngOnInit()
  {
    this.stocks = this.getMockStocks();
  }

  private getMockStocks(): Stock[]
  {
    let stocks: Stock[] = [];

    stocks.push(new Stock('BA', 'Boeing'));
    stocks.push(new Stock('CAT', 'Caterpillar'));
    stocks.push(new Stock('KO', 'Coca-Cola'));

    return stocks;
  }
}
```

Open app.component.html and replace all with the code bellow.

```
<div class="panel panel-warning">

  <div class="panel-heading"><h2>Available stocks</h2></div>

  <table class="table table-striped">
    <tr>
      <th>Symbol</th>
      <th>Company</th>
      <th>Price</th>
    </tr>
    <tr *ngFor="let stock of stocks" >
      <td>{{stock.getSymbol()}}</td>
      <td>{{stock.getCompany()}}</td>
      <td>{{stock.getPrice()}}</td>
    </tr>
  </table>
</div>
```

Here we draw simple table with 3 columns to display information about the stocks. Using bootstrap css classes to make it look nice. Also we use ***ngFor** directive from Angular to generate table rows.

Now go to the browser to look at the results.

It looks good but bootstrap css doesn't work. So you should open index.html and add stylesheet.

```
<link rel="stylesheet"
      href="https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css">
```

We get this css from the internet to make it faster.

Now the app should look much better.

Add random price generator and call it from **getPrice()**.

```
private getRoundedPrice(): number
{
  return Math.round((Math.random() * 1000 * this.symbol.length) * 100 +
      Number.EPSILON) / 100;
}
```

Step 3. Market Component

Let's move market screen to separate component and make it available from app component.

Go to **app** folder and generate new component using angular-cli command:

ng g component market

Now when you open **app.module.ts** you should see MarketComponent inside declarations array.

```
import { MarketComponent } from './market/market.component';

@NgModule({
  declarations: [
    AppComponent,
    MarketComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

Copy html from **app.component.html** to **market.component.html**

Move all the code from **app.component.ts** to **market.component.ts** and leave **AppComponent** class body blank.

```
export class AppComponent { }
```

Open app.component.html and add code bellow.

```
<div style="margin: 50px 50px 50px 50px;">
  <h1>LIVE TRADING</h1>
  <br><br>
  <nav>
    <a class="btn btn-default">Market</a>
    <a class="btn btn-default">Traders</a>
  </nav>
  <br><br>
  <app-market></app-market>
</div>
```

Note: tag **<app-market>** will be replaced with **market.component.html** code according to **templateUrl**.

```
@Component({
  selector: 'app-market',
  templateUrl: './market.component.html',
  styleUrls: ['./market.component.css']
})
export class MarketComponent implements OnInit
```

Run the application. You should see Available Stocks table with **Market** and **Trading** buttons above.

Step 4. Market Editor

Open **market.component.html** and put simple form to add new stock on top.

```
<div class="form-inline">
  <div class="form-group">
    <label for="symbol">Symbol: </label>
    <input #symbol type="text" class="form-control" id="symbol" placeholder="APPL">
  </div>
  <div class="form-group" style="margin-left: 20px;">
    <label for="company">Company: </label>
    <input #company type="text" class="form-control" id="company"
           placeholder="Apple">
  </div>
  <button class="btn btn-default" (click)="add(symbol.value, company.value);
                    symbol.value=''; company.value = ''">Add Stock</button>
</div>
```

Here you can find two attribute directives **#symbol** and **#company**. These directives will force Angular to create variables with the content of specified **<input>** tag. We use it to access the input values inside **(click)** directive.

Open **market.component.ts** and method to process button click event.

```
add(symbol: string, company: string)
{
  this.stocks.push(new Stock(symbol, company));
}
```

Now go to the app and try to add new stocks.

Step 5. Traders Component

Now when Market works fine lets do the Traders screen.

Add new class **Trader.ts** to **domain** directory.

```
export class Trader
{
  constructor(private name: string) { }

  getName(): string
  {
    return this.name;
  }

  getReleasedPnL(): number
  {
    return this.getRoundedPnL(0);
  }

  getUnreleasedPnL(): number
  {
    return this.getRoundedPnL(0);
  }

  getTotalPnL(): number
  {
    return this.getRoundedPnL(this.getReleasedPnL() + this.getUnreleasedPnL());
  }

  private getRoundedPnL(pnl): number
  {
    return Math.round(pnl * 100 + Number.EPSILON) / 100;
  }
}
```

Go to **app** folder and generate new component using angular-cli command:

ng g component traders

Now when you open **app.module.ts** you should see TraderComponent inside declarations array.

Open **app.component.html** and replace **<app-market>** tag with **<app-traders>**. For now we have no ability to switch screens, so will use this hack to check our work.

If you run application you will have the text **traders works!** bellow the buttons.

```
export class TradersComponent implements OnInit
{
  traders: Trader[];

  ngOnInit()
  {
    this.traders = this.getMockTraders();
  }

  private getMockTraders(): Trader[]
  {
    let traders: Trader[] = [];

    traders.push(new Trader('Oleg'));
    traders.push(new Trader('Anna'));

    return traders;
  }
}
```

Go to **traders.component.ts** and add some mock traders.
Open **traders.component.html** and put the code with Traders table.

```
<div class="panel panel-warning">

  <div class="panel-heading"><h2>Traders</h2></div>

  <table class="table table-striped">
    <tr>
      <th>Name</th>
      <th>Unreleased PnL</th>
      <th>Released PnL</th>
      <th>Total PnL</th>
    </tr>
    <tr *ngFor="let trader of traders" >
      <td>{{trader.getName()}}</td>
      <td>{{trader.getUnreleasedPnL()}}</td>
      <td>{{trader.getReleasedPnL()}}</td>
      <td>{{trader.getTotalPnL()}}</td>
    </tr>
  </table>
</div>
```

Note: here we use ***ngFor** structural directive to generate table row for every trader inside **traders** array.

Open **traders.component.html** and put simple form to add new stock on top.

```
<div class="form-inline">
  <div class="form-group">
    <label for="name">Name: </label>
    <input #name type="text" class="form-control" id="name" placeholder="Oleg">
  </div>
  <button class="btn btn-default" (click)="add(name.value); name.value='';">Add
          Trader</button>
</div>

<br><br>
```

Here you can find attribute directive **#name**. This directive will force Angular to create variable with the content of specified **<input>** tag. We use it to access the input value inside **(click)** directive.

```
add(name: string)
{
  this.traders.push(new Trader(name));
}
```

Open **traders.component.ts** and method to process button click event.

Now go to the app and try to add new traders.

Step 6. Routing

It is time to add some navigation and make those two buttons work. We will create **separate angular module** for routing and import it to our application.

Go to **app** folder and generate new module using angular-cli command:

ng g module routing

Configure **routes** array and provide it as root to **RouterModule**. Also we should add our module to the **exports** array to be able to import it inside our **AppModule**.

```
import {RouterModule, Routes} from "@angular/router";
import {MarketComponent} from "../market/market.component";
import {TradersComponent} from "../traders/traders.component";

const routes: Routes =
[
  { path: '', redirectTo: '/market', pathMatch: 'full' },
  { path: 'traders', component: TradersComponent },
  { path: 'market', component: MarketComponent }
];

@NgModule({
  imports: [
    CommonModule,
     RouterModule.forRoot(routes)
  ],
  exports: [RouterModule],
  declarations: []
```

Open **AppModule** and add created **RoutingModule** to **imports** array.

Add `<base href="/">` to index.html after <body> open tag.

Open **app.component.html** and replace **<app-traders>** tag with **<router-outlet>**. This tag will be replaced with html template from specified component according to browser URL.

Start the app you should see Market screen now.

Now let's make our navigation buttons clickable.

Add **routerLink** and **routerLinkActive** attribute directives to the **Market** and **Traders** links. These directives will tell Angular where to navigate when click this link.

```
<h1>LIVE TRADING</h1>
<br><br>
<nav>
  <a class="btn btn-default" routerLink="/market" routerLinkActive="active">Market</a>
  <a class="btn btn-default" routerLink="/traders" routerLinkActive="active">Traders</a>
</nav>
```

Check navigation, should work now and **Market** button should be active when go to root path.

Step 7. Traders Service

Our application will grow up and more components will need access to shared data. So we create single reusable service and inject it into the components that need it.

Using a separate service keeps components lean and focused on supporting the view, and makes it easy to unit-test components with a mock service.

Because data services are invariably asynchronous, you'll create Promise-based version of the data service.

Go to **traders** folder and generate new service using angular-cli command:

ng g service traders

Now you have **traders.service.ts** where the class marked with **@Injectable** annotation.

Open **AppModule** and add created **TradersService** to **providers** array to make it available for injection.

Copy all the code from **traders.component.ts** to the service and add method to get all traders.

Note: for **getTraders()** method we return special **Promise** object and using **setTimeout(…)** to emulate server call.

```
@Injectable()
export class TradersService
{
  traders: Trader[];

  constructor()
  {
    this.traders = this.getMockTraders();
  }

  private getMockTraders(): Trader[]
  {
    let traders: Trader[] = [];

    traders.push(new Trader('Oleg'));
    traders.push(new Trader('Anna'));

    return traders;
  }

  add(name: string)
  {
    this.traders.push(new Trader(name));
  }

  getTraders(): Promise<Trader[]>
  {
    return new Promise(resolve =>
      setTimeout(() => resolve(Promise.resolve(this.traders)), 100));
  }
}
```

Now we can inject this service to **TradersComponent** via constructor and use it to get data.

```
export class TradersComponent implements OnInit
{
  traders: Trader[];

  constructor(private tradersService: TradersService) { }

  ngOnInit()
  {
    this.traders = [];
    this.tradersService.getTraders().then(result => this.traders = result);
  }

  add(name: string)
  {
    this.tradersService.add(name);
  }
```

Note: After component initialized we will get data from service and fill **traders** array. Also we changed **add()** method to make service call.

Now application should work fine with one exception. Traders table will not be updated after we add new trader.

Add **updateTraders()** method to **TradersComponent** and use it when you need data from the server.

```
private updateTraders()
{
  this.tradersService.getTraders().then(result => this.traders = result);
}
```

Check application logic, it should work as before.

Step 8. Market Service

Go to **market** folder and generate new service using angular-cli command:

ng g service market

Now you have **market.service.ts** where the class marked with **@Injectable** annotation.

Open **AppModule** and add created **MarketService** to **providers** array to make it available for injection.

Copy all the code from **MarketComponent** to the service and add method to get all traders.

```
@Injectable()
export class MarketService {

  stocks: Stock[];

  constructor() { this.stocks = this.getMockStocks(); }

  private getMockStocks(): Stock[]
  {
    let stocks: Stock[] = [];

    stocks.push(new Stock('BA', 'Boeing'));
    stocks.push(new Stock('CAT', 'Caterpillar'));
    stocks.push(new Stock('KO', 'Coca-Cola'));

    return stocks;
  }

  getStocks(): Stock[]
  {
    return this.stocks;
  }

  add(symbol: string, company: string)
  {
    this.stocks.push(new Stock(symbol, company));
  }
}
```

Inject **MarketService** to **MarketComponent** and transfer all calls to the service.

```
export class MarketComponent implements OnInit
{
  stocks: Stock[];

  constructor(private marketService: MarketService)
  {
    this.stocks = [];
  }

  ngOnInit()
  {
    this.updateStocks();
  }

  private updateStocks()
  {
    this.stocks = this.marketService.getStocks();
  }

  add(symbol: string, company: string)
  {
    this.marketService.add(symbol, company);
    this.updateStocks();
  }
}
```

Now application should work as usual but if you open Chrome Console window you will see strange error.

ExpressionChangedAfterItHasBeenCheckedError: Expression has changed after it was checked. Previous value: 'xxx'. Current value: ‘yyy'.

This happen because we always return random prices for stocks and Angular doesn’t know what price to show on UI.

Step 9. Price Fetcher

Here we will simulate continues price updates via **MarketService** and make those prices ticking on UI.

We will replace our **MarketService** class with the interface. So open **MarketService** and add interface with 4 public methods after class closing braces.

```
export interface MarketService
{
  getPrice(symbol: string): number;
  getUpdatedPrice(currentPrice: number): number;
  getStocks(): Stock[];
  addStock(symbol: string, company: string);
}
```

Rename the class and add interface Implementation.

```
@Injectable()
export class MarketServiceImpl implements MarketService { ... }
```

Update constructor of **MarketComponent** to get **MarketServiceImpl**.

Finally implement **getPrice()** and **getUpdatedPrice()** methods.

```
getPrice(symbol: string): number
{
  return Math.random() * 1000 * symbol.length;
}

getUpdatedPrice(currentPrice: number): number
{
  let multiplier = 1;
  this.counter++;

  if (this.counter % 2 == 0)
  {
    multiplier = -1;
  }

  return Math.round((currentPrice + (Math.random() * multiplier))
           * 100 + Number.EPSILON) / 100;
}
```

To make **getUpdatedPrice()** work add private variable **counter**.

Add method **getRoundedPrice()** to the **MarketServiceImpl** and update **getPrice()** to use it.

```
private getRoundedPrice(symbol: string): number
{
  return Math.round((Math.random() * 1000 * symbol.length)
           * 100 + Number.EPSILON) / 100;
}
```

Now time to update **Stock** class.

Add **MarketService** interface as constructor parameter.

Implement **initPriceFetcher()** method that will update price every second. For this implementation we will reinitialize the price if it will go bellow zero.

Remove **getRoundedPrice()** method.

Also you should update every **Stock** constructor call inside MarketServiceImpl with service reference.

```
export class Stock
{
  private price: number;

  constructor(private symbol: string, private company: string,
              private marketService: MarketService)
  {
    this.price = this.marketService.getPrice(symbol);
    this.initPriceFetcher();
  }

  private initPriceFetcher()
  {
    setInterval(() =>
    {
      this.price = this.marketService.getUpdatedPrice(this.price);
      if (this.price <= 0)
      {
        this.price = this.marketService.getPrice(this.symbol);
      }
    }, 1000);
  }

  getSymbol(): string
  {
    return this.symbol;
  }

  getCompany(): string
  {
    return this.company;
  }

  getPrice(): number
  {
    return this.price;
  }
}
```

Not it is time to check the application logic. You should see price ticks in real time.

Try to add new stock to the market and make sure it's price tick as well.

Additional tasks

1. Add ability to remove stocks from the market.

Day 4

Step 10. HttpClient

HttpClient provides a simplified API for HTTP functionality for use with Angular applications, building on top of the **XMLHttpRequest** interface exposed by browsers.

Before you can use the HttpClient, **you need to install the HttpClientModule** which provides it. This can be done in your application module, and is only necessary once.

```
@NgModule({
  ...,
  imports: [
    BrowserModule,
    RoutingModule,
    HttpClientModule
  ],
  ...
})
export class AppModule { }
```

In our application we will read the data from the **market-date.json** file provided as static resource from our server. In real world you will get this json data from the remote server.

Open **MarketServiceImpl** and add new interface at the bottom.

```
interface MarketData
{
  symbol: string,
  company: string
}
```

Replace **getMockStocks()** method with getStockData(). This method will return special Observable object that will provide us with the data record by record when it is available.

```
private getStockData(): Observable<MarketData[]>
{
  return this.httpClient.get<MarketData[]>('assets/market-data.json');
}
```

Inject **HttpClient** to constructor and subscribe to **MarketData**.
Run and check the application, you should see all stocks from the file available via market screen.

```
constructor(private httpClient: HttpClient)
{
  this.stocks = [];

  this.getStockData().subscribe(
    data =>
    {
      for (let md of data)
      {
        this.stocks.push(new Stock(md.symbol, md.company, this));
      }
    },
    error =>
    {
      console.log('Cannot get market data from the server!!!');
    }
  );
}
```

Step 11. Trader Details Component

It is time to add trades and we start with Trade object that will store all information about the trade, calculate trade profit and loss in real time, and provide it to the component.

Trader will come and **open Trade** for example buy 10 shares of CAT at $10 per share. This trade will become open **until trader come back again and close it** this some price let's say $12. In this case he will have $2 profit for every share, total 10 shares * $2 = $20. It is called **Released PnL**. Until that time he will have **Unreleased PnL** that will be calculated based on current stock price.

Create new class **Trade** inside domain.

mark - stock price when trade opened
last - last price of the stock
closePrice - stock price when trade closed

Also we will provide reference to the **Stock** object to get actual stock price.

Add **close()** method.

```
close(closePrice: number): void
{
  this.closePrice = closePrice;
  this._isOpen = false;
}
```

```
export class Trade
{
  private mark: number;
  private last: number;

  private _isOpen: boolean;
  private closePrice: number;

  constructor(private stock: Stock, private count: number,
                                                  priceToBuy: number)
  {
    this._isOpen = true;
    this.mark = priceToBuy;
  }

  getStockInfo(): string
  {
    return `${this.stock.getSymbol()} ${this.stock.getCompany()}`;
  }

  getMark(): number
  {
    return this.mark;
  }

  get isOpen(): boolean
  {
    return this._isOpen;
  }

  getClosePrice(): number
  {
    return !this._isOpen ? this.closePrice : 0;
  }

  getCount(): number
  {
    return this.count;
  }

  getStock(): Stock
  {
    return this.stock;
  }
}
```

Add methods:

getUnreleasedPnL() - returns PnL if trade is opened and 0 if closed.
getReleasedPnL() - returns PnL if trade is closed and 0 if opened.
getPnL() - returns PnL in both cases.

Try to implement these methods yourself. Do not forget to round the numbers to have 2 decimal points.

Check your code and make sure it returns same results as code bellow.

```
getUnreleasedPnL(): number
{
  if (!this._isOpen)
  {
    return 0;
  }

  this.last = this.stock.getPrice();
  let tradePnL =  (this.last - this.mark) * this.count;

  return this.getRoundedNumber(tradePnL);
}

getReleasedPnL(): number
{
  if (!this.closePrice)
  {
    return 0;
  }
  let tradePnL =  (this.getClosePrice() - this.mark) * this.count;
  return this.getRoundedNumber(tradePnL);
}


getPnL(): number
{
  let tradePnL =  (this.closePrice - this.mark) * this.count;
  return this.getRoundedNumber(tradePnL);
}

private getRoundedNumber(num: number) : number
{
  return Math.round(num * 100 + Number.EPSILON) / 100;
}
```

Go to **app** folder and generate new component using angular-cli command:

ng g component trader-details

Now when you open **AppModule** you should see **TraderDetailsComponent** inside declarations array.

Open **RoutingModule** and add one more route to the **routes** array.

```
{ path: 'traders/details/Oleg', component: TraderDetailsComponent }
```

Run application, go to **http://localhost:4200/traders/details/Oleg** you will have the text **trader-details works!** bellow the buttons.

Now you sure that new component set up properly.

We start with **Trader** add portfolio array and methods that will calculate PnL for all trades.

```
export class Trader
{
  private portfolio: Trade[];

  constructor(private name: string)
  {
    this.portfolio = [];
  }

  addToPortfolio(trade: Trade)
  {
    this.portfolio.push(trade);
  }

  getPortfolio(): Trade[]
  {
    return this.portfolio;
  }
}
```

Add and implement next methods:

getOpenTrades(): Trade[] - returns all open trades for this trader

getClosedTrades(): Trade[] - returns all closed trades for this trader

getReleasedPnL(): number - returns PnL from all closed trades

getUnreleasedPnL(): number - returns PnL from all **non-closed** trades.

getTotalPnL(): number - returns PnL from all trades together.

Check your code and make sure it returns same results as code bellow.

```
getOpenTrades(): Trade[]
{
  return this.portfolio.filter(trade => trade.isOpen);
}

getClosedTrades(): Trade[]
{
  return this.portfolio.filter(trade => !trade.isOpen);
}

private getRoundedPnL(pnl): number
{
  return Math.round(pnl * 100 + Number.EPSILON) / 100;
}

getReleasedPnL(): number
{
  let pnl = 0;

  for(let trade of this.portfolio)
  {
      pnl += trade.getReleasedPnL();
  }

  return this.getRoundedPnL(pnl);
}

getUnreleasedPnL(): number
{
  let pnl = 0;

  for(let trade of this.portfolio)
  {
    if (trade.isOpen)
    {
      pnl += trade.getUnreleasedPnL();
    }
  }

  return this.getRoundedPnL(pnl);
}

getTotalPnL(): number
{
  return this.getRoundedPnL(this.getReleasedPnL() + this.getUnreleasedPnL());
}
```

We need the method to find trader by name, so implement it inside TradersService. This method will return special **Promise** object that works as proxy and return the trader when found one.

```
getTrader(name: string): Promise<Trader>
{
  return new Promise(resolve =>
  {
      setTimeout(() =>
 resolve(Promise.resolve(this.traders.find(t => name === t.getName()))), 0);
  });
}
```

Inject **TradersService** to **TraderDetailsComponent** and make trader **Oleg** available to UI. Here we use Promise.then function that will be called by Angular if this trader available in the list. You can use **.catch()** method to process the exception if needed.

Note: Create new empty Trader in class constructor to avoid UI binding exceptions.

```
export class TraderDetailsComponent implements OnInit {

  trader: Trader;

  constructor(private tradersService: TradersService)
  {
    this.trader = new Trader('');
  }

  ngOnInit()
  {
    this.tradersService.getTrader('Oleg').then(trader => this.trader = trader);
  }
```

Replace trader-details.component.html content with the code bellow. It will create tables to display portfolio and PnL using methods we created before.

Check the application now. Add some mock trades to portfolio to make sure it works as expected.

Step 12. Buy / Sell the Stock

We want to restrict trader to do anything with the price except reading. So we will add **buyStock()** method to **MarketServiceImpl** that will create and return **Trade**.

```
buyStock(symbol: string, count: number): Trade
{
  let stock: Stock = this.getStock(symbol);

  if (stock)
  {
    return new Trade(stock, count, stock.getPrice());
  }

  return null;
}

private getStock(symbol: string): Stock
{
  return this.stocks.find(stock => stock.getSymbol() == symbol);
}
```

Now we need to provide stock symbol somehow… Let's do it with **Angular Reactive forms**.

Open **AppModule** and add **ReactiveFormsModule** to the **imports** array.

Add two local variables **countInput** and **symbolInput** to **TraderDetailsComponent** we will use it to access the data from html form.

Then add **buyStock()** to **TraderDetailsComponent**

```
buyStock()
{
  let trade: Trade =
this.marketService.buyStock(this.symbolInput.value,
this.countInput.value);

  if (!trade)
  {
    alert(`symbol ${this.symbolInput.value} not found`);
    return;
  }

  this.trader.addToPortfolio(trade);

  this.symbolInput.setValue('');
}
```

```
<div class="row">
  <div class="col-md-4">
    <h1>{{trader.getName()}}</h1>
    <h2>Unreleased PnL: {{trader.getUnreleasedPnL()}}</h2>
    <h2>Released PnL:   {{trader.getReleasedPnL()}}</h2>
    <h2>Total PnL:      {{trader.getTotalPnL()}}</h2>
  </div>
  <div class="col-md-4">

  </div>
  <div class="col-md-4">

  </div>
</div>

<br><br>
<div class="row">
  <div class="col-md-8">
    <div class="panel panel-warning">

      <div class="panel-heading"><h2>Open positions</h2></div>

      <table class="table table-striped">
        <tr>
          <th>#</th>
          <th>Symbol</th>
          <th>Mark</th>
          <th>Count</th>
          <th>Unreleased PnL</th>
          <th></th>
        </tr>
        <tr *ngFor="let trade of trader.getOpenTrades(); let i = index;" >
          <td>{{i}}</td>
          <td>{{trade.getStockInfo()}}</td>
          <td>{{trade.getMark()}}</td>
          <td>{{trade.getCount()}}</td>
          <td>{{trade.getUnreleasedPnL()}}</td>
          <td><button class="btn btn-default">Close</button></td>
        </tr>
      </table>
    </div>

    <div class="panel panel-warning">

      <div class="panel-heading"><h2>Closed Trades</h2></div>

      <table class="table table-striped">
        <tr>
          <th>#</th>
          <th>Symbol</th>
          <th>Mark</th>
          <th>Close Price</th>
          <th>Count</th>
          <th>PnL</th>
        </tr>
        <tr *ngFor="let trade of trader.getClosedTrades(); let i = index;" >
          <td>{{i}}</td>
          <td>{{trade.getStockInfo()}}</td>
          <td>{{trade.getMark()}}</td>
          <td>{{trade.getClosePrice()}}</td>
          <td>{{trade.getCount()}}</td>
          <td>{{trade.getPnL()}}</td>
        </tr>
      </table>
    </div>

  </div>
  <div class="col-md-4">
    <div>

    </div>
  </div>
</div>
```

Open component html template and code bellow to the second column after Total PnL.

```
<div class="form-group">
  <label>Symbol</label>
  <input type="text" class="form-control" id="symbol" [formControl]="symbolInput"  >
</div>
<div class="form-group">
  <label for="count">Count</label>
  <input type="number" class="form-control" id="count" [formControl]="countInput"  >
</div>
<button class="btn btn-success" (click)="buyStock()" id="buyStockBtn" >Buy Stock</button>
```

Note: we use **[formControl]** syntax to bind html input field to appropriate component variable. Also **(click)** syntax to bind the method.

Check the app, add some stocks to portfolio and make sure grids below and PnL updated.

Add close button after Unreleased PnL to every trade from Open Trades table.

```
<td><button class="btn btn-default" (click)="closeTrade(trade)">Close</
button></td>
```

Add **closeTrade()** method implementation to **MarketServiceImpl** and call it from **TraderDetailsComponent#closeTrade(Trade)**.

```
sellStock(trade: Trade): void
{
  let stock: Stock = trade.getStock();
  trade.close(stock.getPrice());
}
```

Check the application, add new trades, close some of them and make sure everything works fine.

Step 13. Trader Details Navigation

It is time to include new screen to our routing and make it work with all traders.

Open routing module and replace hardcoded route **traders/details/Oleg** path with **traders/details/:name**

This syntax allow us to read name from url and use it to find needed Trader.

Open TraderDetailsComponent and inject **ActivatedRoute** to the component constructor. Then update **ngOnInit()** to read trader name when url has changed.

```
constructor(…, private route: ActivatedRoute)

ngOnInit(): void
{
  this.route.paramMap
    .switchMap((params: ParamMap) =>
this.tradersService.getTrader(params.get('name')))
    .subscribe((trader: Trader) => this.trader = trader);
}
```

Note: **route.paramMap** contains name-value pairs where name - parameter name we specified in route path and value - test from url (in our case trader's name). Also we use **swithMap** operator from **rxjs library** to track url change. We need to import this operator manually.

```
import 'rxjs/add/operator/switchMap';
```

Now it is time to create references for every trader. Open **traders.component.html** and replace trader name with the link.

```
<td><a routerLink="/traders/details/{{trader.getName()}}"
routerLinkActive="active">
{{trader.getName()}}</a></td>
```

Note: **routerLink** directive provides angular with url.

Open trader-details.component.html and add Back button right after the traders name.

```
<button class="btn btn-primary" (click)="goBack()">Back</button>
```

To make it work we should do 2 things. Inject **Location** service from **@angular/common** to the **TraderDetailsComponent** and implement goBack() method that will transfer call to injected location.

```
constructor(…, private location: Location)

goBack(): void
{
  this.location.back();
}
```

Now feel free to test the application.

Step 14. Stocks component

Now the only thing I do not like in this application it's symbol input field from traders details form. It is much better to have something like **Drop down where you can choose from of the existing stocks with search filter** and this step will replace that input.

We replace symbol input field with new component. So go to trader-details and generate new stocks component.

ng g component stocks

Now when you open **AppModule** you should see **StocksComponent** inside declarations array.

This component will contains a list of stocks from where we will select one at a time.

Angular team have one more interesting project for us. It's name Angular Material and they already have control called Autocomplete. You can find it here https://material.angular.io/components/autocomplete/overview

To setup this component we have to add next modules to **AppModule** imports array.

```
import { MatAutocompleteModule, MatInputModule, MatOptionModule } from
"@angular/material";

@NgModule({
  declarations: [
     ...
  ],
  imports: [
   ...,
    MatInputModule,
    MatAutocompleteModule,
    MatOptionModule,
    BrowserAnimationsModule
  ],
  providers: [...],
  bootstrap: [...]
})
```

And because these modules come from external library we should download it from remote repository first. As you already know all needed libs will be downloaded according to **package.json** file, so update **dependencies** array and add two more libraries.

```
"@angular/cdk": "^2.0.0-beta.8",
"@angular/material": "^2.0.0-beta.8"
```

Run **npm install** to build the project.

Replace symbol input field with <app-stocks> </app-stocks> tag inside **trader-details.compontnt.html**.

Run the app and make sure it works fine.

Now we will add this Autocomplete control to the stocks html template and provide it with the list of stocks to select from.

Inject **MarketServiceImpl** to the component constructor to get stocks from.
For Autocomplete to work we need **stockInput** as field to insert symbol and **filteredStocks** array to put filtered data from **stocks** array. Here we use specific Observable object that would allow us to change values when **stockInput.value** changes.

Open component template and copy html bellow.
Note: Here we basically describe how to get and display values from autocomplete then connect it to input field using matAutocomplete name.

Also add link to angular material css to your **styles.css** file
Now you can check the app, it will display input field, you can type something into it but it will not work for now.

```
@import '~@angular/material/prebuilt-themes/deeppurple-amber.css';
```

```
<mat-form-field>
  <input type="text" matInput
[formControl]="stockInput" [matAutocomplete]="auto" id="stockInput" >
</mat-form-field>

<mat-autocomplete #auto="matAutocomplete">
  <mat-option *ngFor="let stock of filteredStocks |
async" [value]="stock.getSymbol()">
```

```
export class StocksComponent implements OnInit {

  stockInput = new FormControl();
  filteredStocks: Observable<Stock[]>;

  stocks: Stock[];
```

This field should display filtered list of stock symbols based on **filteredStocks** array so let's initialize it.

```
ngOnInit()
{
  this.stocks = this.marketService.getStocks();

  this.filteredStocks = this.stockInput.valueChanges
    .startWith(null)
    .map(val => val ? this.filter(val) : this.stocks.slice());
}

filter(val: string): Stock[] {
  return this.stocks.filter(stock => new RegExp(`^${val}`, 'gi')
                                        .test(stock.getSymbol()));
}
```

We will fill this array from scratch every time **stockInput** value changes.

Now run the application and try to type something you should see the list of available stocks.

We will use *ngIf structural directive to display company name for selected symbol.

Open stock component html template and add code bellow.

```
<div class="form-group">
  <label *ngIf="selected != null">Company: {{selected.getCompany()}}</label>
</div>
```

Create one more subscription to **stockInput.value** updates that will look for specified symbol and update **selected** stock.

Run the application and check that it works as expected.

```
ngOnInit()
{

...

this.stockInput.valueChanges.startWith(null).subscribe(symbol =>
  {
      let stock = this.findStock(symbol);
      if (stock != null)
      {
        this.selected = stock;
      }
  });
}

findStock(symbol: string): Stock
{
  return this.stocks.find(stock => symbol === stock.getSymbol());
}
```

When stock selected we should send this selected stock to parent component **TraderDetailsComponent**. Angular has 7 ways to do that. In this case we will use the way where Parent component listens for child event.

Open **StocksComponent** and add event as new field.

```
import {EventEmitter} from '@angular/core';

@Output()
onStockSelect = new EventEmitter<Stock>();
```

Update **stockInput.value** change subscriber to also emit the event after set up selected stock.

```
this.onStockSelect.emit(stock);
```

This code will emit event from the **StocksComponent** when user select any existing stock.

To subscribe open **TraderDetailsComponent** add method to set selected stock, also create a local variable for that.

```
selectedStock: Stock;

onStockSelect(stock: Stock)
{
  this.selectedStock = stock;
}
```

Open trader-details.component.html and update <app-stocks> tag.

```
<app-stocks (onStockSelect)="onStockSelect($event)" ></app-stocks>
```

This syntax will bind event emitter to the **onStockSelect()** method from **TraderDetailsComponent**.

Now we already have selected stock and it is time to update **buyStock()** method logic.

```
buyStock()
{
  if (this.selectedStock == null)
  {
    window.alert('Please select the stock');
    return;
  }

  let trade: Trade =
          this.marketService.buyStock(this.selectedStock.getSymbol(),
                                      this.countInput.value);
  this.trader.addToPortfolio(trade);

  this.selectedStock = null;
}
```

Delete useless **symbolInput** field and run the application. Add some stocks to portfolio for every trader and make sure the logic works fine.

It would be great to clean symbol field after we buy the stock and for that we need to access child component **StocksComponent**. Here we will use another way for component interaction it is called - Parent calls an @ViewChild().

Add method clean to the **StocksComponent**.

```
clean()
{
  this.stockInput.setValue('');
  this.selected = null;
}
```

Open **TraderDetailsComponent** and add new field.

```
@ViewChild(StocksComponent)
private stocksComponent: StocksComponent;
```

This code force angular to inject the link to child **StocksComponent**. And last step that we need to do - call created **clean()** method after buying the stock.

Congratulations!!! You are done with the core functions. Run the app check.

Step 15. Unit (jusmine) testing

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM.

Go and run the tests

Run **npm run test**

You should have compilation error. If you look at the market.service.spec.ts you will have **MarketService** as provider but we rename it to **MarketServiceImpl** and now you trying to test the interface.

Replace **MarketService** with **MarketServiceImpl** and tests should run.

8 of 9 tests failed it is ok for now, because we have updated the code but didn't update the tests. Will do it one by one.

The simplest will be independent unit tests for the services. We will start with **TradersService**. It contains one generated test that check service creation and **it is only one green** for us now.

TEST: Check that service created using **toBeDefined** method.

All the other methods from **TradersService** return promises and we should do a little tricks to test it. For now I want we to learn the basics so open **TradersService** and add methods that returns us current traders array.

```
getTradersInstant(): Trader[]
{
  return this.traders;
}
```

TEST: Check that mock traders created using **toBeDefined** method.

TEST: Check that we have created exactly 2 mock traders using **toEqual** method.

TEST: Check that 1st trader name is **Oleg** using **toEqual** method.

TEST: Check that 2nd trader name is **Anna** using **toEqual** method.

TEST: Check that we have created exactly 2 mock traders using **toEqual** method.

TEST: Check **add** method.

Now test methods that return **Promise** objects. In this case you should use different syntax for **it** method.

```
it('message', function(done)
{
    ...
    done();

}, timeout in ms);
```

TEST: Check **getTrader** method it returns Promise instantly so just check the object and call **done()** right after the **expectation**.

TEST: Do same thing (check **getTrader** method) with usual **it function** (no timeout) and make sure it is fail.

TEST: Check **getTraders** method it returns Promise instantly so just check the object and call **done()** right after the **expectation**.

TEST: Do same thing (check **getTraders** method) with usual **it function** (no timeout) and make sure it is fail.

Test **MarketServiceImpl** class.

This class uses **Angular Http** module and we cannot just create the object like we did for **TradersService**. We will let Angular **to do the injection** and then test it.

Update beforeEach() method, add imports array and put HttpModule there.

```
let marketService: MarketServiceImpl;

beforeEach(() =>
{
  TestBed.configureTestingModule({
    imports: [HttpClientModule],
    providers: [MarketServiceImpl]
  });

  marketService = TestBed.get(MarketServiceImpl);
});
```

Note: TestBed it is a method to produce the module environment for the class you want to test. In effect, you detach the tested component from its own application module and re-attach it to a dynamically-constructed Angular test module tailored specifically for this battery of tests.

We will talk about this later, just use it for now.

Now you can use **marketService** variable to access configured service.

TEST: Check that service created using **toBeDefined** method.

Or use Angular **inject** function.

The inject function is one of the Angular testing utilities. It injects services into the test function where you can alter, spy on, and manipulate them.

The inject function has two parameters:

1. An array of Angular dependency injection tokens.
2. A test function whose parameters correspond exactly to each item in the injection token array.

TEST: Check that service created **using inject function** and check that stocks array also created.

```
it('should contain stocks', inject([MarketServiceImpl],
                                   (service: MarketServiceImpl) =>
{
  expect(service.getStocks()).toBeDefined();
}));
```

TEST: Check that stocks array contains 30 stocks after initialization. **Note:** Since we use http client to get stocks it takes some time to do it. So, do not forget to do the timeout. It is a little bit tricky because we do not use Promise here but I believe you can make it.

TEST: Check that stocks array contains GE stock after initialization.

TEST: Check **addStock** method.

If you look at the **Stock** class it takes **MarketService** as constructor parameter and then use it to fetch the price. Here we **need to check Stock class** and do not care how MarketService works, so we will mock it.

Create new **stock.spec.ts** file and put it to domain folder.

Initial code will be:

```
describe('Stock Isolated Tests with MarketService spy', () =>
{
  let marketServiceSpy: MarketService;

  beforeEach(() =>
  {
    marketServiceSpy =
    {
      getPrice(symbol: string): number { return 0; },
      getUpdatedPrice(currentPrice: number): number { return 0; },
      getStocks(): Stock[] { return [] },
      addStock(symbol: string, company: string) { }
    };

  });
});
```

Then we use **jasmine spyOn** method to create expected behavior to **getPrice(...)** and **getUpdatedPrice(...)** methods.

```
spyOn(marketServiceSpy, 'getPrice');
spyOn(marketServiceSpy, 'getUpdatedPrice');
```

TEST: Check Stock object creation. Use **toHaveBeenCalledTimes()** method to make sure you call getPrice exactly one time.

TEST: Check Stock object creation. Use **toHaveBeenCalledWith()** method to make sure you call getPrice exact symbol you pass as constructor parameter.

TEST: Check that you call getUpdatedPrice every second from Stock object creation.

Step 16. Unit (jusmine) testing, more practice.

Create tests for all public methods on two classes Trade and Trader.

Step 17. Angular Testing Tools

It is time to test our components. Will use detach the tested component from its own application module and re-attach it to a dynamically-constructed Angular test module tailored specifically for this battery of tests.

Will start with the easiest **StocksComponent**. It has **MarketServiceImpl** as provider and we should mock it here.

Create testing folder and put new **market.service.spy.spec.ts** inside.

```
export class MarketServiceSpy implements MarketService
{
  getStocks = jasmine.createSpy('getStocks')
          .and.callFake(() => this.getFakeMStocks());

  getPrice = jasmine.createSpy('getPrice')
          .and.callFake(() => 100);

  getUpdatedPrice = jasmine.createSpy('getUpdatedPrice')
          .and.callFake(() => 110);

  buyStock(symbol: string, count: number): Trade
  {
    let stock: Stock = new Stock(symbol, '',
          new MarketServiceSpy());

    return new Trade(stock, count, stock.getPrice());
  }

  addStock(symbol: string, company: string)
  {
  }

  private getFakeMStocks(): Stock[]
  {
    let stocks: Stock[] = [];

    stocks.push(new Stock('MMM', '3M', this));
    stocks.push(new Stock('MCD', "McDonald's", this));
    stocks.push(new Stock('MRK', 'Merck', this));
    stocks.push(new Stock('MSFT', 'Microsoft', this));

    return stocks;
  }
```

Note: We need getStocks() method only for this test, let it return some fake stocks.

Edit **async beforeEach** method and add providers array with **MarketServiceImpl** service. Also add **overrideComponent** block to use **created spy object** instead of real service.

```
beforeEach(async(() =>
{
  TestBed.configureTestingModule({

    declarations: [StocksComponent],
     providers: [MarketServiceImpl]
  })
    .overrideComponent(StocksComponent,
      {
        set: {
          providers: [{provide: MarketServiceImpl,
                       useClass: MarketServiceSpy}]
        }
      }
     )
    .compileComponents();
}));
```

Now add list of imports for autocomplete component from Angular Material library.

```
imports: [
  BrowserAnimationsModule,
  MatInputModule,
  MatAutocompleteModule,
  MatOptionModule,
  ReactiveFormsModule
]
```

Run the tests. Test component should be created.

TEST: Check that **stockInput** field exists. Use **fixture.debugElement.query(…)** to get element from the template.

TEST: Check that stock selected according to existing filter. Update **stockInput** field and check **selected** field.

TEST: Check that selected field become null if filter returns 0 stocks. Note this test should not pass because we have a little bug. Fix it and make test green.

TEST: Check that company label is shown and contains valid company name when the stock selected. Use **fixture.detectChanges()** to update html bindings and **fixture.debugElement.query(…)** to get label value.

```
it('company label should be displayed when stock selected', () =>
{
  let filter = 'MMM';
  let companyName = '3M';

  component.stockInput.setValue(filter);

  fixture.detectChanges();

  let companyLabel: DebugElement =
    fixture.debugElement.query((de: DebugElement) => de.name === 'label');
  expect(companyLabel.nativeElement.innerText).toContain(companyName);
});
```

TEST: Check that company label is hidden when no stock selected.

Our **TraderDetailsComponent** using **ActivatedRoute** and **Location** objects that should be mocked for the tests.

Go to testing folder and add new file **router-stubs.spec.ts** for router stubs. Our main goal is **paramMap** because we use it inside component.

```
@Injectable()
export class ActivatedRouteStub
{
  private subject = new Subject();
  paramMap = this.subject.asObservable();

  private _testParamMap: Params;

  get testParamMap()
  {
    return this._testParamMap;
  }

  set testParamMap(params: Params)
  {
    this._testParamMap = params;
    this.subject.next(convertToParamMap(this._testParamMap));
  }

  get snapshot()
  {
    return { paramMap: this.testParamMap };
  }
}
```

Also create a spy for LocationStrategy with empty methods. Will use separate file **location.strategy.spy.spec.ts** for that.

```
export class LocationStrategySpy
{
  back = jasmine.createSpy('back').and.callFake(() => {});

  getBaseHref = jasmine.createSpy('getBaseHref').and.callFake(() => "/");
  path = jasmine.createSpy('back').and.callFake(() => {});
  prepareExternalUrl = jasmine.createSpy('prepareExternalUrl').and.callFake(() => {});
  pushState = jasmine.createSpy('pushState').and.callFake(() => {});
  replaceState = jasmine.createSpy('replaceState').and.callFake(() => {});
  forward = jasmine.createSpy('forward').and.callFake(() => {});
  onPopState = jasmine.createSpy('onPopState').and.callFake(() => {});

}
```

Now go and set up component tests.

We will need and instance of our ActivatedRout and expected trader name as global variables.

```
let expectedTraderName: string = 'Oleg';
let activatedRoute: ActivatedRouteStub = new ActivatedRouteStub();
```

Set up async beforeEach to override component providers.

```
beforeEach(async(() =>
{
  TestBed.configureTestingModule({
    imports: [
      BrowserAnimationsModule,
      MatInputModule,
      MatAutocompleteModule,
      MatOptionModule,
      ReactiveFormsModule
    ],
    declarations: [TraderDetailsComponent, StocksComponent],
    providers: [
      TradersService,
      {provide: MarketServiceImpl, useClass: MarketServiceSpy},
      {provide: ActivatedRoute, useValue: activatedRoute},
      Location,
      {provide: LocationStrategy, useClass: LocationStrategySpy},
    ]
  })
    .compileComponents();
}));
```

Add out trader name to the **testParamMap** inside **beforeEach()**.

```
activatedRoute.testParamMap = {name: expectedTraderName};
```

Run the tests to check everything setup correctly.

TEST: Check **trader** setup after component initialization. Use **it** function with timeout, we need some time for component initialization before check.

TEST: Check **buyStock** functionality. Use **it** function with timeout, and **fixture.debugElement.query()** to get **buyStockBtn** and do the click.

```
it('stock should be added to portfolio', function (done)
{
  let expectedStockSymbol = 'XXX';

  setTimeout(() =>
  {
    component.selectedStock = new Stock(expectedStockSymbol, '',
                                        new MarketServiceSpy());
    component.countInput.setValue(10);

    let buyStockBtn: DebugElement = fixture.debugElement.query(
            (de: DebugElement) => de.attributes['id'] === 'buyStockBtn');
    buyStockBtn.nativeElement.click();

    let actualTrade = component.trader.getPortfolio()
      .find(trade => expectedStockSymbol === trade.getStock().getSymbol());

    expect(actualTrade.getStock().getSymbol()).toEqual(expectedStockSymbol);

    done();

  }, 100)

}, 200);
```

Create tests for **MarketComponent**, use **MarketServiceSpy**.

TEST: Check component created.

TEST: Check add stock functionality.

Note: Update **MarketServiceSpy** to support add stock, use **fixture.debugElement.query()** to get elements from html.

Our TradersComponent contains router links that targets specific trader details screen. To test this component we will replace Angular **routerLink** directive with out stub.

Open **router-stubs.spec.ts** and add new directive.

```
@Directive({
  selector: '[routerLink]',
  host: {
    '(click)': 'onClick()'
  }
})
export class RouterLinkStubDirective
{
  @Input('routerLink')
  linkParams: any;

  navigatedTo: any = null;

  onClick() {
    this.navigatedTo = this.linkParams;
  }
}
```

Go to component tests and add **RouterLinkStubDirective** to declarations array inside **async beforeEach** method.

Also set up **TradersService** as provider and create global variable for it.

Run the tests and check component created properly.

TEST: Check html template contains input with id name.

TEST: Check html template contains a button.

TEST: Check html template contains 2 links with our mock traders Oleg and Anna. Look for routerLinkActive attribute using **fixture.debugElement.queryAll()** method.

Note: Make sure you query for links after **fixture.detectChanges()** call, because you have no traders on component init stage because of async call from **updateTraders()**.

TEST: Check add trader functionality from component side. Add new trader via **tradersService** and check if it is displayed on html after **fixture.detectChanges()**.

TEST: Check add trader functionality from html side. Set name, click a button and then check if new trader added.

The last component from tests will be out **AppComponent**. This component does basically nothing but contains <router-outlet> tags and links to our screens.

To test this component we need to stubs: **RouterLinkStubDirective** and **RouterOutletStubComponent**. We already have first one, so open **router-stubs.spec.ts** and add next code:

```
@Component({selector: 'router-outlet', template: ''})
export class RouterOutletStubComponent { }
```

Now go and set up component test.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [
      AppComponent,
      RouterLinkStubDirective,
      RouterOutletStubComponent
    ],
  })
    .compileComponents()
    .then(() =>
    {
      fixture = TestBed.createComponent(AppComponent);
      component = fixture.debugElement.componentInstance;
    });
}));
```

Note: Here we use **.then** function to initialize objects for test. It is just one another way to do it.

TEST: Check that component created and works well.

Add next **beforeEach()** method.

```
beforeEach(() =>
{
  fixture.detectChanges();

  links = fixture.debugElement.queryAll(By.directive(RouterLinkStubDirective));
});
```

Note: We can also use directive class in queries.

TEST: Check that we have only two links on the page.

TEST: Check that we have link to the /market page.

TEST: Check that we have link to the /traders page.

Congratulations!!! You are done. Now you should understand how to test Angular 4 application.

Check out the docs https://angular.io/guide/testing#testing to get more information about this topic.