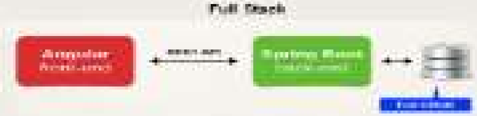
Full Stack Project Overview

Project

- Build Real-time eCommerce App

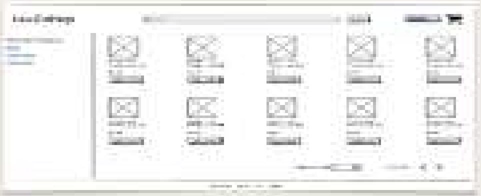


Requirements

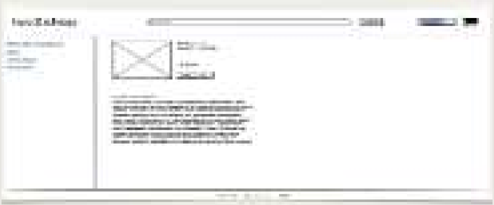
[illegible]

- Show a list of products
- Add products to shopping cart (CRUD)
- Shopping cart check out
- User login/logout security
- Track previous orders for logged in users

Wireframes - Home Page



Wireframes - Product Details



Wireframes - Shopping Cart Details

Wireframes - Check Out

Release Plan

- Release 1.0
  - Show a list of products
- Release 2.0
  - Add products to shopping cart (CRUD)
  - Shopping cart check out
- Release 3.0
  - User login/logout security
  - Track previous orders for logged in users

Spring Boot Back End

Java Development Environment

- We assume that you are already an experienced Spring Boot Developer

- You should have the following items already installed
  - Java Development Kit (JDK)
  - Java IDE (we'll use IntelliJ in the videos, but any Java IDE will work)
  - Maven
  - MySQL Database and MySQL Workbench

About IntelliJ

Super Amazing IDE!!!

- In this course, we will use the free version of IntelliJ
  - Known as IntelliJ Community Edition
  - Download from: **https://www.jetbrains.com/idea/download**
  - **Select Community Edition**

- You can also use the Ultimate Edition ($) free trial version is available

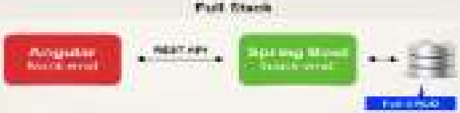
Additional Java IDEs

- You are free to use other Java IDEs such as Eclipse, VS Code, NetBeans
- All you need is a Java IDE that supports Maven … that's it!
- You can easily follow along with any Java IDE

- We will provide tech support for IntelliJ, Eclipse, VS Code, NetBeans

Spring Boot Back End

- Leverage Spring Data REST for REST API
- Minimizes the coding for Spring Boot back end



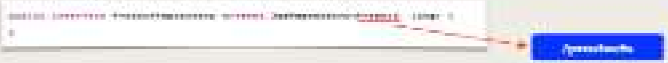
Create Repository

- Spring Data REST will scan your project for JpaRepository
- Expose REST APIs for each entity type for your JpaRepository

```
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

REST Endpoints

- By default, Spring Data REST will create endpoints based on entity type
    - Simple pluralized form
        - First character of Entity type is lowercase
        - Then just adds an "s" to the entity



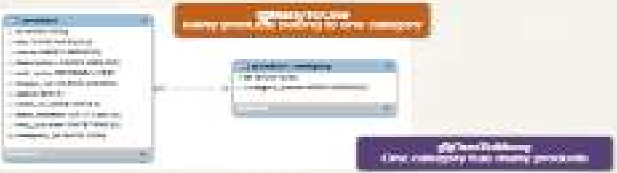
```
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

REST API

Spring Data REST will expose these endpoints for free!

HTTP Method		CRUD Action
POST	/products	Create a new product
GET	/products	Read a list of products
GET	/products/{id}	Read a single product
PUT	/products/{id}	Update an existing product
DELETE	/products/{id}	Delete an existing product

Database Schema - Release 1.0



@ManyToOne
Many products pointing to one category

@OneToMany
One category has many products

Two Database Scripts

- 01-create-user.sql
- 02-create-products.sql

About: 01-create-user.sql

1. Create a new MySQL user for our application
   - user id: ecommerceapp
   - password: ecommerceapp

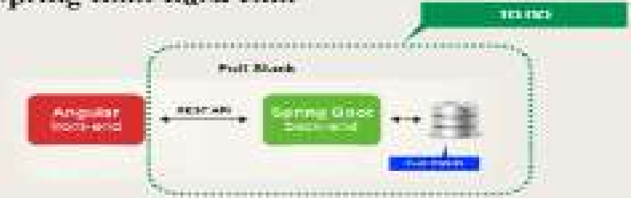
About: 02-create-products.sql

1. Create new database tables: product, product_category

2. Load tables with sample data

- full-stack-ecommerce
  - Tables
    - product
    - product_category
  - Views
  - Stored Procedures
  - Functions

Spring Boot Back End



Development Process

Step-By-Step

1. Set up the database tables
2. Create a Spring Boot starter project (start.spring.io)

```
spring-boot-starter-data-jpa
spring-boot-starter-data-rest
mysql-connector-java
 lombok
```

3. Develop the Entities: Product and ProductCategory
4. Create REST APIs with Spring Data JPA Repositories and Spring Data REST

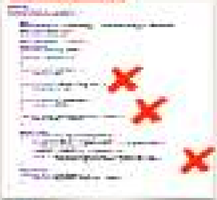
Project Lombok

- Modern Java project
- Lombok automagically generates the getters / setters (behind the scenes)
- No need for the developer to manually define getters / setters, etc ...
- Easy-to-use Annotations to eliminate boilerplate code

http://www.projectlombok.org

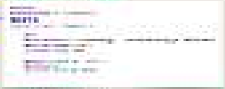
Project Lombok

Before Lombok



After Lombok

Lombok annotation



That's it!!
Absolutely no need to generate getters and setters

Lombok will do the work for you automagically behind the scenes

REST API - Read Only

REST API

I don't want that!
I want the REST API as READ-ONLY

- Spring Data REST will expose ALL these endpoints for free✗?

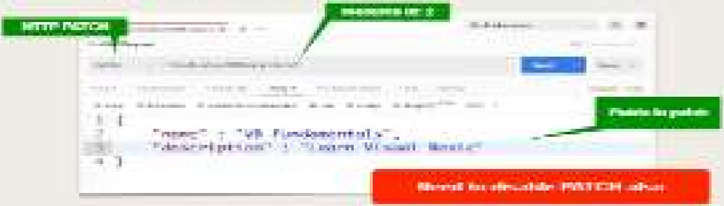
HTTP Method		CRUD Action
✗ POST	/products	Create a new product
✓ GET	/products	Read a list of products
✓ GET	/products/{id}	Read a single product
✗ PUT	/products/{id}	Update an existing product
✗ DELETE	/products/{id}	Delete an existing product

REST API: PATCH

- Spring Data REST also exposes an endpoint to accept PATCH requests
- HTTP method: PATCH is used to perform partial updates on a resource
  - Provide the resource id in the URL path
  - List the fields to "patch"
  - Will update the "patch" fields, all other fields remain the same

Need to disable PATCH also

HTTP PATCH Example



```
{
    "name" : "VB fundamentals",
    "description" : "Learn Visual Basic"
}
```

Possible Solutions

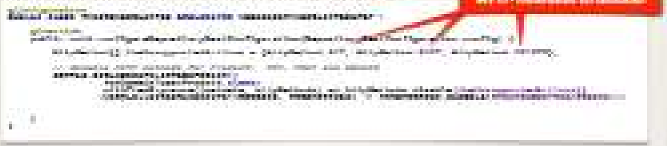
1. Option 1: Don't use Spring Data REST
   1. Manually create our own @RestController
   2. Manually define methods for Access: @GetMapping
   3. But we lose the Spring Data REST support for paging, sorting etc :-(

2. Option 2: Use Spring Data REST — Choose this one!
   1. **Configure to disable certain HTTP methods: POST, DELETE etc**

We will cover REST API Security later in the course

Spring Data REST Configuration

- Disable HTTP methods: POST, PUT, DELETE

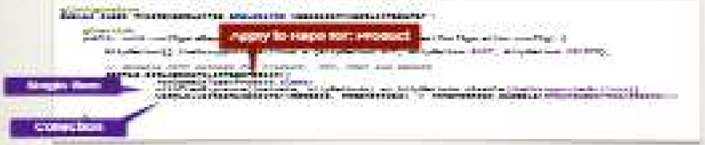


```java
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {

        HttpMethod[] theUnsupportedActions = {HttpMethod.PUT, HttpMethod.POST, HttpMethod.DELETE};

        // disable HTTP methods for Product: PUT, POST and DELETE
        config.getExposureConfiguration()
                .forDomainType(Product.class)
                .withItemExposure((metdata, httpMethods) -> httpMethods.disable(theUnsupportedActions))
                .withCollectionExposure((metdata, httpMethods) -> httpMethods.disable(theUnsupportedActions));
    }
}
```

HTTP methods to disable

Spring Data REST Configuration

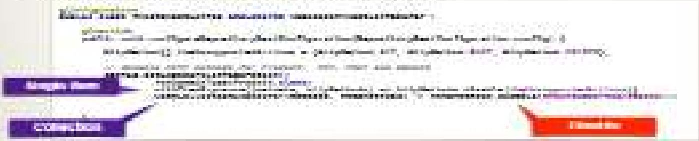
- Disable HTTP methods: POST, PUT, DELETE



```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {

        HttpMethod[] theUnsupportedActions = {HttpMethod.PUT, HttpMethod.POST, HttpMethod.DELETE};

        // disable HTTP methods for Product: PUT, POST and DELETE
        config.getExposureConfiguration()
                .forDomainType(Product.class)
                .withItemExposure((metdata, httpMethods) -> httpMethods.disable(theUnsupportedActions))
                .withCollectionExposure((metdata, httpMethods) -> httpMethods.disable(theUnsupportedActions));
```

Spring Data REST Configuration

- Disable HTTP methods: POST, PUT, DELETE



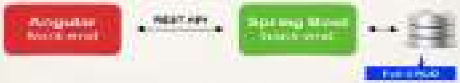
```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {

    @Override
    public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config) {

        HttpMethod[] theUnsupportedActions = {HttpMethod.PUT, HttpMethod.POST, HttpMethod.DELETE};

        // disable HTTP methods for Product: PUT, POST and DELETE
        config.getExposureConfiguration()
                .forDomainType(Product.class)
                .withItemExposure((metdata, httpMethods) -> httpMethods.disable(theUnsupportedActions))
                .withCollectionExposure((metdata, httpMethods) -> httpMethods.disable(theUnsupportedActions));
```

Angular Front End

Angular Front End

- Create Angular Front End components
- Retrieve data from Spring Boot REST APIs

Full Stack



Development Process

Step-By-Step

1. Create Angular project
2. Create Angular component for product-list
3. Develop TypeScript class for Product
4. Create Angular service to call REST APIs
5. Update Angular component to subscribe to data from Angular service
6. Display the data in an HTML page
7. Add CrossOrigin support to Spring Boot app

Step 1: Create Angular project

- Create new project using Angular CLI

```
C:\> ng new angular-ecommerce
```

Step 2: Create Angular component for product-list

- Create new component using Angular CLI

```
$ ng generate component components/product-list
```

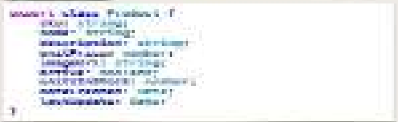
Generated files placed in sub-directory:
components/product-list

Step 3: Develop TypeScript class for Product

- Create new class

Placed in sub-directory: common

```
C:\> ng generate class common/product
```

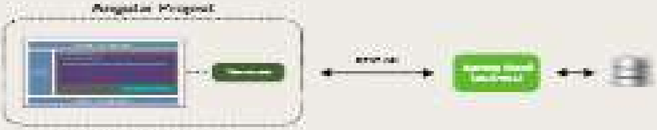


```
export class Product {
    sku: string;
    name: string;
    description: string;
    unitPrice: number;
    imageUrl: string;
    active: boolean;
    unitsInStock: number;
    dateCreated: Date;
    lastUpdated: Date;
}
```

Step 4: Create Angular service to call REST APIs

- Angular "Service" is code developed in TypeScript
- Service is a helper class that provides desired functionality
- Part of your Angular application and runs in the web browser client-side

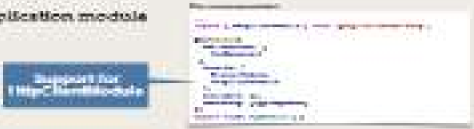
Application Interaction



Runs in Web Browser
Client Side

Step 4: Create Angular service to call REST API

- REST client provided by Angular
  - HttpClient ... part of HttpClientModule
- Add support in the application module



Support for HttpClientModule

Step 4: Create Angular service to call REST APIs

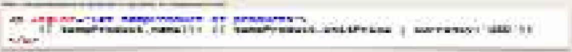
Step 5: Develop Angular to subscribe to data

```
[illegible]
```

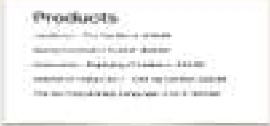
[illegible]

[illegible]

Step 6: Display the Data in an HTML page



```
<li *ngFor="let tempProduct of products">
    {{ tempProduct.name }} - {{ tempProduct.unitPrice | currency:'USD' }}
</li>
```



Products

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Step 7: Add CrossOrigin support to Spring Boot

[illegible]

- By default, this coding will fail
- Web browsers will not allow script code to call APIs not on same origin
- Known as Same-origin policy
- Same origin is composed of: scheme/protocol, hostname, port number
- Can relax this by adding "Cross-Origin Resource Sharing (CORS)" on server side application

Add CrossOrigin support to Spring Boot App

[illegible]

```
@CrossOrigin("http://localhost:4200")
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

[illegible]

```
@CrossOrigin({"http://localhost:4200", "http://www.mycoolapp.com"})
```

[illegible]

```
@CrossOrigin
```

Online Store Template Integration

Release 2.0 - Plan

- Online Shop Template Integration
- Search for products by category
- Search for products by text box
- Master / detail view of products
- Pagination support for products
- Add products to shopping cart (CRUD)
- Shopping cart check out

[illegible]

Current task

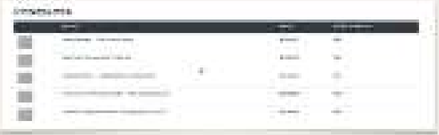
More functionality than originally planned with 2.0

"Feature creep"

[illegible]

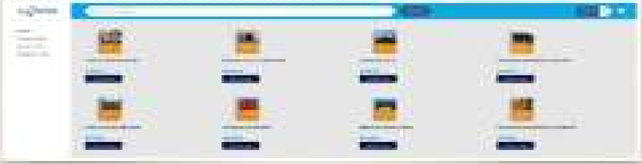
Current Status

- At the moment, we have a basic front end.
- Proof of concept for listing the products: Angular + Spring Boot



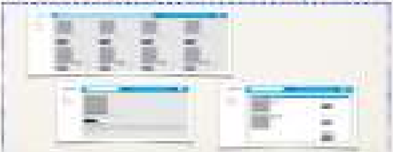
Online Shop Template Integration

- We need a website that looks like an online shop



Wireframes to Web Template

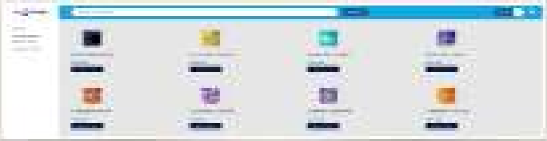
- I sent the wireframes to a friend.
- He created a web template using HTML and CSS



He delivered a zip file *.html and *.css

Online Shop Template Integration

- Integrate the HTML template with our Angular + component .html files
- We'll add the CSS styles to our project



A Note about Cascading Style Sheets (CSS)

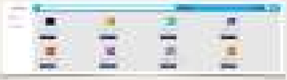
- The template uses CSS for styles, layout and colors
- Teaching CSS is beyond the scope of this course
- If you would like a tutorial on CSS, free online tutorials are available

www.w3schools.com/css

www.w3schools.com/bootstrap4

Development Process

1. Download the HTML template starter files
2. Install Bootstrap CSS styles locally using npm
3. Add local custom CSS styles to Angular src/styles.css file
4. Integrate template files into Angular app
5. Add support for menu and logos
6. Enhance our application with product images



Search By Category

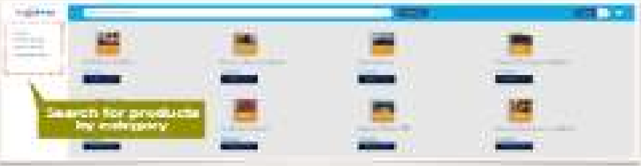
Release 2.0 - Plan

[illegible]

- Online Shop Template Integration
- Search for products by category
- Search for products by text box
- Master / detail view of products
- Pagination support for products
- Add products to shopping cart (CRUD)
- Shopping cart check out

Current task

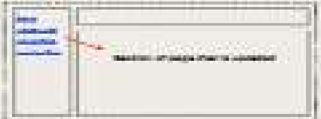
Search for Products by Category



Search for products by category

Angular Routing

- In Angular, you can add links in your application
- The links will route to other components in your application
- Angular routing will handle updating a view of your application



Only updates a section of your view
Doesn't reload entire page

Key Components

Name	Description
Router	[illegible]
Route	Maps a URL path to a component.
RouterOutlet	[illegible]
RouterLink	Link to specific routes in your application.
ActivatedRoute	The current active route that loaded the component. Useful for accessing route parameters.
Params	

http://angular.io/guide/router

[illegible]

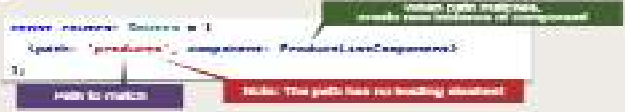
Development Process

Step-By-Step

1. Define routes
2. Configure Router based on our routes
3. Define the Router Outlet
4. Set up Router Links to pass category id param
5. Enhance ProductListComponent to read category id param
6. Modify Spring Boot app - REST Repository needs new method
7. Update Angular Service to call new URL on Spring Boot app

Step 1: Define Routes

- A route has a path and a reference to a component
- When the user selects the link for the route path
  - Angular will create a new instance of component



```
const routes: Routes = [
  {path: 'products', component: ProductListComponent}
];
```

When path matches,
create new instance of component

Path to match

Note: The path has no leading slash!

Step 1: Define Routes

- Add route to show products for a given category id

```
const routes: Routes = [
  {path: 'category/:id', component: ProductListComponent},
  {path: 'products', component: ProductListComponent}
];
```

Category as parameter

The component can read this value and show products for this category

Step 1: Define Routes

- Add more routes to handle for other cases ...

```
const routes: Routes = [
  {path: 'category/:id', component: ProductListComponent},
  {path: 'category', component: ProductListComponent},
  {path: 'products', component: ProductListComponent},
  {path: '', redirectTo: '/products', pathMatch: 'full'},
  {path: '**', redirectTo: '/products', pathMatch: 'full'}
];
```

Order of routes is important.
First match wins.
Start from most specific to generic.

This is the generic wildcard. It will match on anything that didn't match above routes.

Step 1: Define Routes

- Can add a custom PageNotFoundComponent … for 404s

```
const routes: Routes = [
...
 {path: '**', component: PageNotFoundComponent}
];
```

[illegible]

Order of routes is important. First match wins.
Start from most specific to generic.

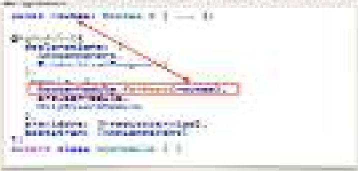
Custom component that you create

Can give any name and provide your own custom view

https://angular.io/guide/router#define-a-wildcard-route

Step 2: Configure Router based on our routes

- Configure the router in the application module:



```
const routes: Routes = [ ... ];

@NgModule({
  declarations: [
    AppComponent,
    [illegible]
  ],
  imports: [
    RouterModule.forRoot(routes),
    BrowserModule,
    [illegible]
  ],
  providers: [illegible],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 3: Define the Router Outlet

- Router Outlet acts as a placeholder
- Renders the desired component based on route.

[illegible]

Step 3: Define the Router Outlet

- Update app.component.html to use Router Outlet

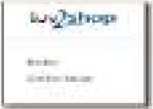
```
<!-- MAIN CONTENT -->
<router-outlet></router-outlet>

<!-- <app-product-list></app-product-list> -->
```

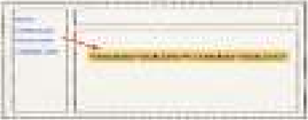
Based on Router configuration
display appropriate component here

Comment out or delete the reference to
<app-product-list></app-product-list>

Step 4: Set up Router Links to pass category id param

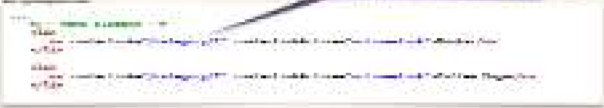
- In our HTML page, set up links to our route
- Pass category id as a parameter





Step 4: Set up Router Links to pass category id param

- In our HTML page, set up links to our route
- Pass category id as a parameter

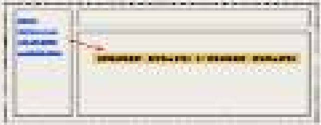


Based on Route configuration, use ProductListComponent

```
...
  <!-- MENU ELEMENTS -->
  <li>
    <a routerLink="/category/1" routerLinkActive="active-link">Books</a>
  </li>

  <li>
    <a routerLink="/category/2" routerLinkActive="active-link">Coffee Mugs</a>
  </li>
```

Recall: Router Outlet

- When user clicks the link
  - ProductListComponent will appear in the location of router-outlet



Only updates a section of your page

Does not reload entire page

Step 5: Enhance ProductListComponent to read category id param

- Need to read the category id parameter

```
currentCategoryId: number;

this.currentCategoryId = +this.route.snapshot.paramMap.get('id');
```

State of route at this given moment in time

Read the id parameter

Use the activated route

Map of all the route parameters

Step 6: Modify Spring Boot app - REST Repository needs new method

- Currently, the Spring Boot app, returns products regardless of category
- Need to modify to only return products for a given category id

Step 6: Modify Spring Boot app - REST Repository needs new method

- Spring Data REST and Spring Data JPA supports "query methods"
- Spring will construct a query based on method naming conventions
- Methods starting with: findBy, readBy, queryBy, etc ...

Magic!

[illegible]

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);
}
```

A query method

Step 6: Modify Spring Boot app - REST Repository needs new method

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);
}
```

A query method

Match by category id

Use this parameter value

Behind the scenes, Spring will execute a query similar to this:

SELECT * FROM product where category_id=?

Magic!

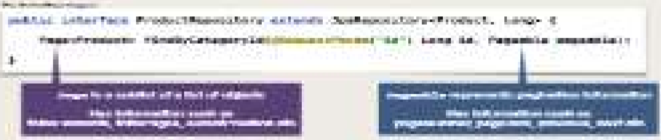
More on Query Methods

- You can provide your own custom query using @Query annotation
- Support is available for conditionals: and, or, like, sort etc
- For details on this topic
  - See the Query Method section in the Spring Data Reference Manual

www.luv2code.com/spring-data-query-methods

Step 6: Modify Spring Boot app - REST Repository needs new method

- Page and Pageable provides support for pagination



ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);

}
```

Page is a sublist of a list of objects

Has information such as totalElements, totalPages, currentPosition etc

Pageable represents pagination information

Has information such as pageNumber, pageSize, previous, next etc

Step 5: Modify Spring Boot app - REST Repository needs new method

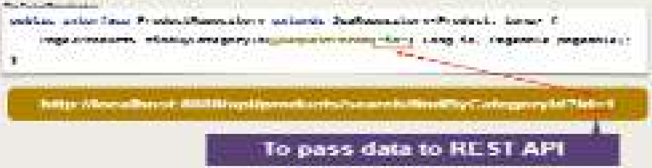
- Spring Data REST automatically exposes endpoints for query methods
- **Exposes endpoint: /search/<<queryMethodName>>**

ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);
}
```

http://localhost:8080/api/products/search/findByCategoryId

Step 6: Modify Spring Boot app - REST Repository needs new method



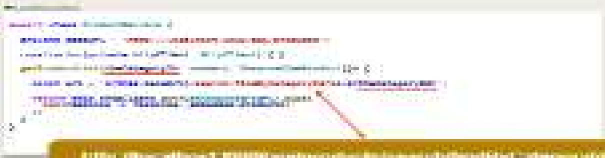
ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByCategoryId(@RequestParam("id") Long id, Pageable pageable);
}
```

http://localhost:8080/api/products/search/findByCategoryId?id=1

To pass data to REST API

Step 7: Update Angular Service to call new URL on Spring Boot app



```
[illegible]
```

http://localhost:8080/api/products/search/findByCategoryId?id=1

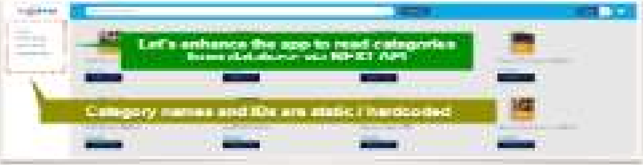
Development Process

Step-By-Step

1. Define routes
2. Configure Router based on our routes
3. Define the Router Outlet
4. Set up Router Links to pass category id param
5. Enhance ProductListComponent to read category id param
6. Modify Spring Boot app - REST Repository needs new method
7. Update Angular Service to call new URL on Spring Boot app

Search By Category

Search for Products by Category



Let's enhance the app to read categories from database via REST API

Category names and IDs are static / hardcoded

Development Process

Step-By-Step

1. Modify Spring Boot app - Expose entity ids
2. Create a class: ProductCategory
3. Create new component for menu
4. Enhance menu component to read data from product service
5. Update product service to call URL on Spring Boot app
6. In HTML, replace hard-coded links with menu component

Step 1: Modify Spring Boot app - Expose entity ids

- By default, Spring Data REST does not expose entity ids
- We need entity IDs for a number of use cases
  - Get a list of product categories by id
  - Master / detail view – get a product by id

Step 1: Modify Spring Boot app - Expose entity ids

- By default, Spring Data REST does not expose entity ids

There is no entity id of the product category item

Actually, entity id is embedded in the HATEOAS links

Still no easy access. Requires parsing URL string.

This approach is cumbersome and brittle :-(

Step 1: Modify Spring Boot app - Expose entity ids

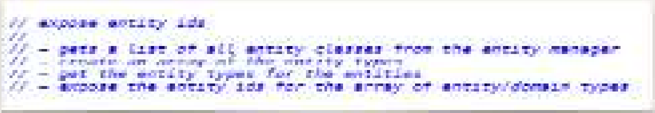
- This is what you need

Entity id at
the product-category level

Entity [illegible] = 1

Step 1: Modify Spring Boot app - Expose entity ids

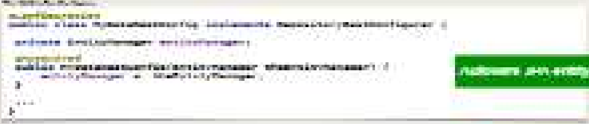
- Update Spring Data REST config to expose entity ids.



```
// expose entity ids
//
// - gets a list of all entity classes from the entity manager
// - create an array of the entity types
// - get the entity types for the entities
// - expose the entity ids for the array of entity/domain types
```

Step 1: Modify Spring Boot app - Expose entity ids

- Update Spring Data REST config to expose entity ids



MyDataRestConfig.java

```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {

  private EntityManager entityManager;

  @Autowired
  public MyDataRestConfig(EntityManager theEntityManager) {
      entityManager = theEntityManager;
  }

  ...
}
```

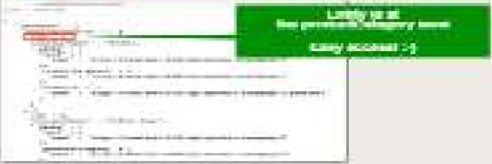
Autowire JPA entity manager

Step 1: Modify Spring Boot app - Expose entity ids

```
[illegible]
```

Step 1: Modify Spring Boot app - Expose entity ids

- The configuration gives us the desired output



Entity id at
the product category level

Easy access :-)

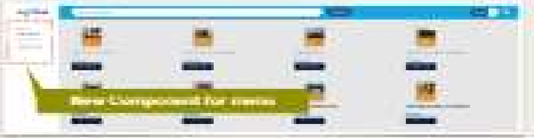
Step 2: Create class: ProductCategory

```
> ng generate class common/product-category
```

[illegible]

```
export class ProductCategory {
    id: number;
    categoryName: string;
}
```

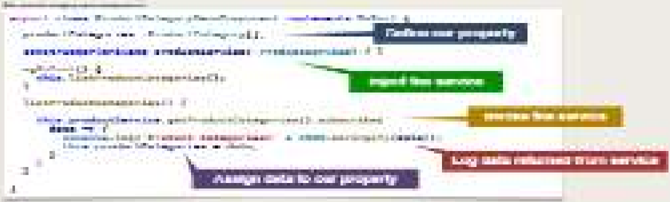
Step 3: Create new component for menu



New Component for menu

```
> ng generate component components/product-category-menu
```

Step 4: Enhance component to read categories from service



```
export class ProductCategoryMenuComponent implements OnInit {
  productCategories: ProductCategory[];
  constructor(private productService: ProductService) { }
  ngOnInit() {
    this.listProductCategories();
  }
  listProductCategories() {
    this.productService.getProductCategories().subscribe(
      data => {
        console.log('Product Categories=' + JSON.stringify(data));
        this.productCategories = data;
      }
    );
  }
}
```

Define our property

Inject the service

Invoke the service

Log data returned from service

Assign data to our property

Step 5: Update product service to call URL on Spring Boot app

```
[illegible]
```

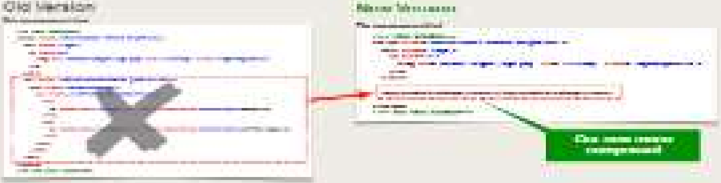
URL for product categories

Call REST API

[illegible]

[illegible]

Step 6: Replace hard-coded links with menu component



Step 6: Replace hard-coded links with menu component

Loop over categories and build links dynamically

Display category name

Pass category id to view

[illegible]

Search By Keyword

Search for Products by Keyword

Search for products by keyword

Development Process

Step-By-Step

1. Modify Spring Boot app - Add a new search method
2. Create new component for search
3. Add new Angular route for searching
4. Update SearchComponent to send data to search route
5. Enhance ProductListComponent to search for products with ProductService
6. Update ProductService to call URL on Spring Boot app

Step 1: Modify Spring Boot app - Search Method

- Spring Data REST and Spring Data JPA supports "query methods"
- Spring will construct a query based on method naming conventions

ProductRepository.java

Magic!

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);
    ...
}
```

Step 1: Modify Spring Boot app - Search Method

- Find products based on name

ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);

}
```

"Containing" … similar to SQL: "LIKE"

Step 1: Modify Spring Boot app - Search Method

[illegible]

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);
```

[illegible]

```
SELECT * FROM Product p
WHERE
p.name LIKE CONCAT('%', :name , '%')
```

Step 1: Modify Spring Boot app - Search Method

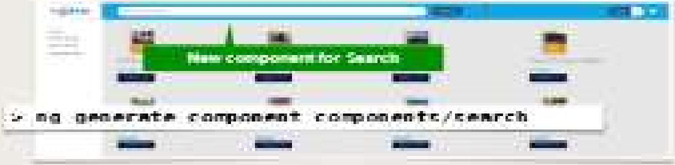
[illegible]

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    Page<Product> findByNameContaining(@RequestParam("name") String name, Pageable pageable);
    ...
}
```

http://localhost:8080/api/products/search/findByNameContaining?name=Python

To pass data to REST API

Step 2: Create new component for Search



New component for Search

```
> ng generate component components/search
```

Step 3: Add new Angular route for searching

File: app.module.ts

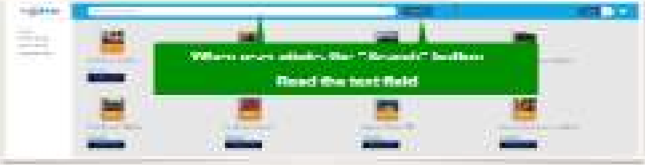
```
{path: 'search/:keyword', component: ProductListComponent},
```

Parameter

Quick Discussion

Event Binding

Quick Discussion - Event Binding

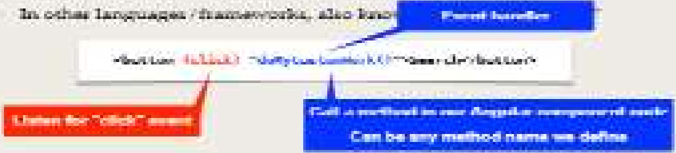


When user clicks the "Search" button

Read the text field

Event Binding

- In Angular, you can listen for events with "event binding"
- In other languages / frameworks, also kno



```
<button (click)="doMyCustomWork()">Search</button>
```

Event handler
Listen for "click" event
Call a method in our Angular component code
Can be any method name we define

Event Binding - Example

Call a method on our Angular component code

```
<button (click)="doMyCustomWork()">Click Me</button>
```

Listen for "click" event

```
export class SearchComponent implements OnInit {
  doMyCustomWork() {
    console.log('Hey! You clicked on button!');
  }

  // ...
}
```

Reading User Input

symbol
Template reference variable

Provides access to the element

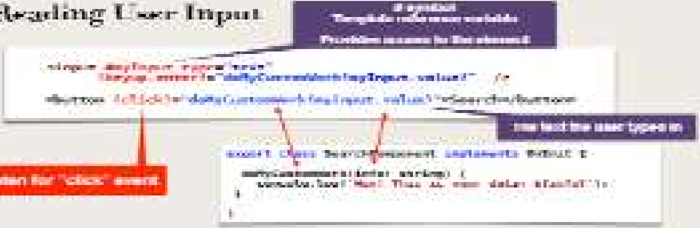
```
<input #myInput type="text"
       (keyup.enter)="doMyCommandWork(myInput.value)" />
```

The text the user typed in

Listen for the "enter" key

```
export class SearchComponent implements OnInit {

  doMyCommandWork(data: string) {
    console.log(`Hey! This is your data: ${data}`);
  }

}
```

Reading User Input



symbol
Template reference variable

Provides access to the element

```
<input #myInput type="text"
       (keyup.enter)="doMyCustomWork(myInput.value)" />

<button (click)="doMyCustomWork(myInput.value)">Search</button>
```

Use text the user typed in

Listen for "click" event

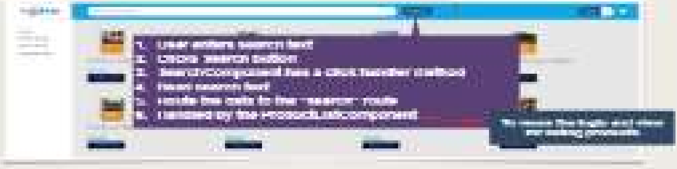
```
export class SearchComponent implements OnInit {

  doMyCustomWork(data: string) {
    console.log(`Hey! This is your data: ${data}`);
  }

}
```

Other Events

Name	Description
focus	An element has received focus
blur	An element has lost focus
keyup	Any key is released. [illegible]
keydown	Any key is pressed
dblclick	The mouse is clicked twice on an element
More	

https://developer.mozilla.org/en-US/docs/Web/Events

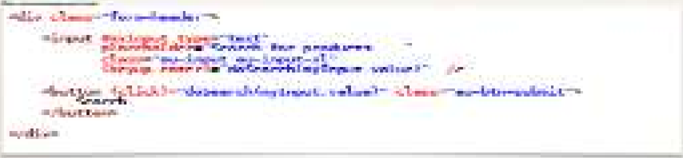
Step 5: Update SearchComponent to send data to search route



1. User enters search text
2. Clicks search button
3. SearchComponent has a click handler method
4. Reset search text
5. Route the data to the "search" route
6. Handled by the ProductListComponent

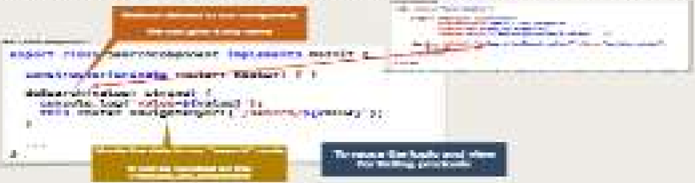
We reuse the logic and view for listing products

Step 5: Update SearchComponent to send data to search route



```
<div class="form-inline">

    <input #myInput type="text"
           placeholder="Search for products ..."
           class="au-input au-input-xl"
           (keyup.enter)="doSearch(myInput.value)" />

    <button (click)="doSearch(myInput.value)" class="au-btn-submit">
        Search
    </button>

</div>
```

Step 5: Update SearchComponent to send data to search route



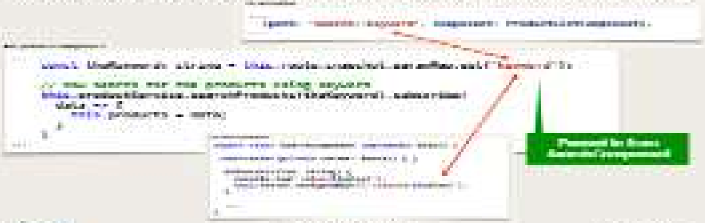
```
export class SearchComponent implements OnInit {

  constructor(private router: Router) { }

  doSearch(value: string) {
    console.log(`value=${value}`);
    this.router.navigateByUrl(`/search/${value}`);
  }

  ...
}
```

[illegible]

[illegible]

[illegible]

Step 5: Enhance component to search for products with product service



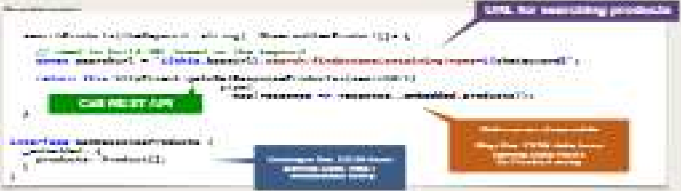
```
{path: 'search/:keyword', component: ProductListComponent},
```

```
const theKeyword: string = this.route.snapshot.paramMap.get('keyword');

// now search for the products using keyword
this.productService.searchProducts(theKeyword).subscribe(
  data => {
    this.products = data;
  }
)
```

```
export class SearchComponent implements OnInit {

  constructor(private router: Router) { }

  doSearch(value: string) {
    console.log(`value=${value}`);
    this.router.navigateByUrl(`/search/${value}`);
  }

  ...
}
```

Passed in from SearchComponent

Step 6: Update product service to call URL on Spring Boot app



```
searchProducts(theKeyword: string): Observable<Product[]> {

  // need to build URL based on the keyword
  const searchUrl = `${this.baseUrl}/search/findByNameContaining?name=${theKeyword}`;

  return this.httpClient.get<GetResponseProducts>(searchUrl).
                        pipe(
                          map(response => response._embedded.products));
}

interface GetResponseProducts {
  _embedded: {
    products: Product[];
  }
}
```

URL for searching products

Call REST API

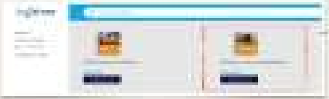
Returns an observable

Map the JSON data from Spring Data REST to Product array

Unwraps the JSON from Spring Data REST _embedded entry

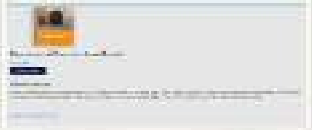
Product Master-Detail View

Master-Detail View



Master View

User clicks the product to see details



Detail View

Application shows the details for selected product

Development Process

Step-By-Step

1. Create new component for product details
2. Add new Angular route for product details
3. Add router links to the product list grid HTML page
4. Enhance ProductDetailsComponent to retrieve product from ProductService
5. Update ProductService to call URL on Spring Boot app
6. Update HTML page for ProductDetailsComponent to display product details

Step 1: Create new component for product details

```
> ng generate component components/ProductDetails
```

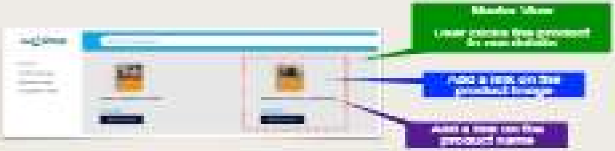
Step 2: Add new Angular route for product details

[illegible]

```
{path: 'products/:id', component: ProductDetailsComponent},
```

Parameter

Step 3: Add router links to product-list-grid HTML



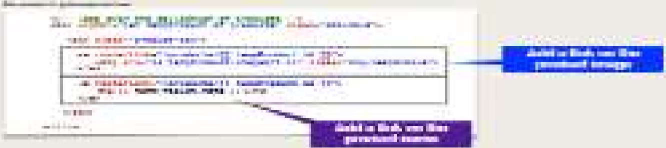
Master View

User clicks the product for more details

Add a link on the product image

Add a link on the product name

Step 3: Add router links to product-list-grid HTML

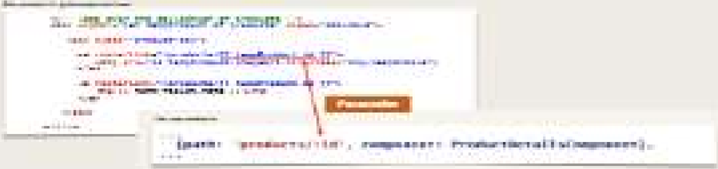


[illegible]

Add a link on the product image

Add a link on the product name

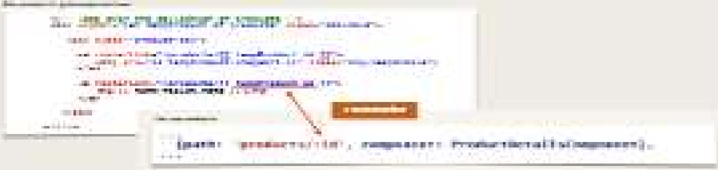
Step 3: Add router links to product-list-grid HTML.



```
[illegible]
```

```
...
{path: 'products/:id', component: ProductDetailsComponent},
...
```

Step 3: Add router links to product-list-grid HTML.



```
[illegible]
```

```
...
{path: 'products/:id', component: ProductDetailsComponent},
...
```

Step 4: Enhance ProductDetailComponent to retrieve product from ProductService

[illegible]

```
...
    // get the "id" param string. convert string to a number using the "+" symbol
    const theProductId: number = +this.route.snapshot.paramMap.get('id');

    this.productService.getProduct(theProductId).subscribe(
      data => {
        this.product = data;
      }
    )
...
```

[illegible]

Parameter

```
  {path: 'products/:id', component: ProductDetailsComponent},
...
```

Step 5: Update product service to call URL on Spring Boot app

product.service.ts

```
getProduct(theProductId: number): Observable<Product> {
  // need to build URL based on product id
  const productDetailsUrl = `${this.baseUrl}/${theProductId}`;
  return this.httpClient.get<Product>(productDetailsUrl);
}
```

URL for retrieving a product

http://localhost:8080/api/products/{theProductId}

Call REST API

Returns an observable

[illegible]

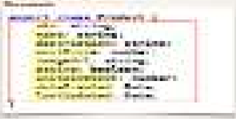
[illegible]

Mapping JSON to Product class

[illegible]

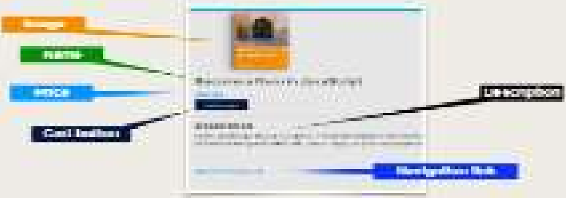
[illegible]

[illegible]

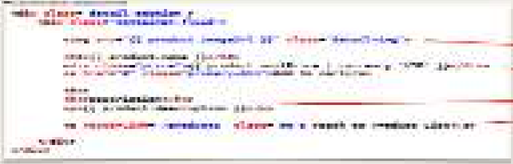


```
export class Product {
    id: string;
    name: string;
    description: string;
    unitPrice: number;
    imageUrl: string;
    active: boolean;
    unitsInStock: number;
    dateCreated: Date;
    lastUpdated: Date;
}
```

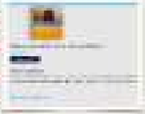
Step 8: Update HTML page for ProductDetailComponent to display product details



Step 8: Update HTML page for ProductDetailComponent to display product details



```
<div class="detail-section">
    <div class="container-fluid">

        <img src="{{ product.imageUrl }}" class="detail-img">

        <h3>{{ product.name }}</h3>
        <div class="price">{{ product.unitPrice | currency:'USD' }}</div>
        <a href="#" class="primary-btn">Add to cart</a>

        <hr>
        <h4>Description</h4>
        <p>{{ product.description }}</p>

        <a routerLink="/products" class="mt-5">Back to Product List</a>

    </div>
</div>
```



Angular Language Service

Angular / JavaScript - Lenient

- Angular / JavaScript is VERY lenient

- Even though we are using TypeScript … some bugs can still slip through!

Angular / JavaScript - Lenient

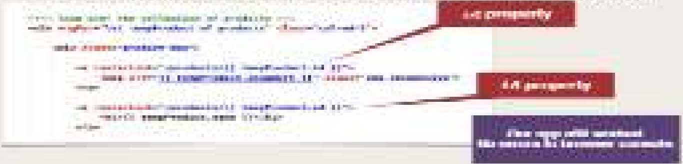
- For example, our product class does not have an id property defined
- But we can STILL assign data to it ... what????

No id property

Our app still worked
No errors in browser console

Angular / JavaScript - Lenient

We also referenced the products .id property for our master / detail view



```
<!-- loop over the collection of products -->
<div *ngFor="let tempProduct of products" class="col-md-3">

    <div class="product-box">

        <a routerLink="/products/{{ tempProduct.id }}">
            <img src="{{ tempProduct.imageUrl }}" class="img-responsive">
        </a>

        <a routerLink="/products/{{ tempProduct.id }}">
            <h1>{{ tempProduct.name }}</h1>
        </a>
```

Our app still worked
No errors in browser console

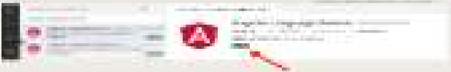
Angular Language Service

- Ideally, we would like to add more compile time checks for our application
- The Angular team provides the Angular Language Service
- Provides additional support for
  - Compilation errors, code completion, etc.

https://angular.io/guide/language-service

Angular Language Service

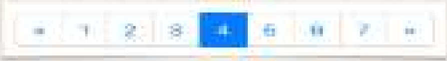
- Available as extensions for code editors: Visual Studio Code, WebStorm etc ...
- Install the extension in Visual Studio Code:
  - Menu option: View -> Extensions
  - Search for: Angular Language Service
  - Click: Install



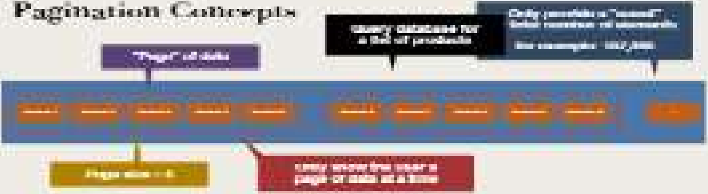
Pagination

Pagination

- Pagination is useful for handling large amounts of data
- Show the users a small subset of data: "page" of data
- The user can click links to view other pages



Pagination Concepts



Query database for a list of products

Only possible if a "small" total number of elements

For example: [illegible]

"Page" of data

Page size = 5

Only show the user a page of data at a time

Pagination

- We will need pagination support on the back-end: Spring Boot
- Also need pagination support on the front-end: Angular

Pagination

- Our Spring Boot backend uses Spring Data REST

- Spring Data REST provides pagination support out of the box … yes!!!

www.luv2code.com/spring-data-rest-pagination

Spring Data REST - Parameters

- By default, Spring Data REST returns 20 elements.
- We can customize this by passing in parameters

Parameter	Purpose
page	The page number to access, 0-based ... defaults to 0.
size	The size of the page to return (items per page). Defaults to 20.

Spring Data REST - Pagination Examples

Get the first page, with page size of 10

```
http://localhost:8080/api/products?page=0&size=10
```

Remember:
Pages are 0 based.
First page is at position 0

Get the second page, with page size of 10

```
http://localhost:8080/api/products?page=1&size=10
```

Second page is at position 1

Spring Data REST - Response Meta Data

The response meta data has valuable information

```
.. array of products ..
...
"page" : {
  "size" : 10,
  "totalElements" : 200,
  "totalPages" : 20,
  "number" : 0
}
```

Size of this page

Total number of ALL elements in the database. But we are NOT returning all of the elements, Just the "count" for informational purposes only

Total pages available

Current page number

Pagination with Angular

- There are many pagination solutions available for Angular
- We will make use of a popular component framework: ng-bootstrap

https://ng-bootstrap.github.io

Bootstrap widgets

[illegible]

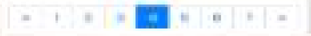
Components in ng-bootstrap

https://ng-bootstrap.github.io/

Pagination component

[illegible]

Parameter	Purpose
page	The page number to display. 1-based -- defaults to 1
pageSize	The size of the page (items per page). Defaults to 10
collectionSize	The total number of items.
pageChange	Event handler for page change events.



Pagination component

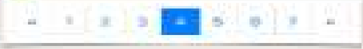
[illegible]

Basic example of pagination component

```
<ngb-pagination [illegible]
                [illegible]
                [illegible]
                [illegible]
</ngb-pagination>
```

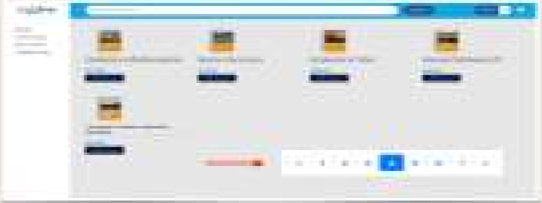
We will create code to calculate links for our project

Component will generate links for pagination



Pagination - Development Process

Pagination



Development Process

1. Install ng-bootstrap
2. Refactor the interface for GetResponseProducts
3. Add pagination support to ProductService
4. Update ProductListComponent to handle pagination
5. Enhance HTML template to use ng-bootstrap pagination component

Step 1: Install ng-bootstrap

- Run the following commands in your Angular project directory

```
> ng add @angular/localize

> npm install @ng-bootstrap/ng-bootstrap
```

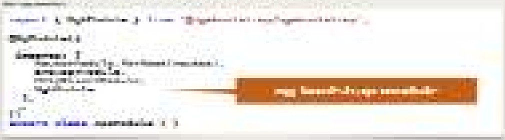
Dependency for Angular 9+

Installs ng-bootstrap

https://ng-bootstrap.github.io/#/getting-started

Step 1: Install ng-bootstrap

- Import the module for ng-bootstrap

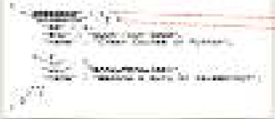


```
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';

@NgModule({

  imports: [
      RouterModule.forRoot(appRoutes),
      BrowserModule,
      HttpClientModule,
      NgbModule
  ],

})
export class AppModule { }
```

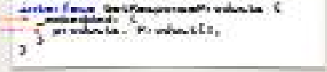
ng-bootstrap module

Step 2: Refactor the interface for GetResponseProducts

- We currently use the interface GetResponseProducts
- Maps JSON data from REST API to our TypeScript objects



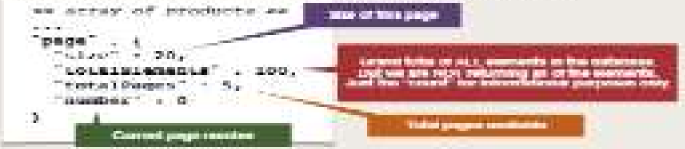
```
{
  "_embedded" : {
    "products" : [ {
      "id" : 1,
      "sku" : "BOOK-TECH-1000",
      "name" : "Crash Course in Python",
      ...
    }, {
      "id" : 2,
      "sku" : "BOOK-TECH-1001",
      "name" : "Become a Guru in JavaScript",
      ...
    } ]
  }
}
```



```
interface GetResponseProducts {
  _embedded: {
    products: Product[];
  }
}
```

Spring Data REST - Response Meta Data

The response meta data also has valuable information for pagination



```
... array of products ...
...
"page" : {
  "size" : 20,
  "totalElements" : 100,
  "totalPages" : 5,
  "number" : 0
}
```

Size of this page

[illegible]

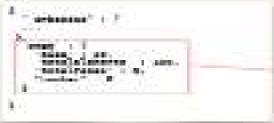
Total pages available

Current page number

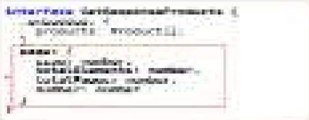
Step 2: Refactor the interface for GetResponseProducts

- Refactor the interface to support the pagination meta data

JSON



```
{
  "_embedded" : {

  "page" : {
    "size" : 20,
    "totalElements" : 100,
    "totalPages" : 5,
    "number" : 0
  }
}
```



```
interface GetResponseProducts {
  _embedded: {
    products: Product[];
  },
  page: {
    size: number,
    totalElements: number,
    totalPages: number,
    number: number
  }
}
```

Step 3: Add pagination support to ProductService

product.service.ts

Pass in parameters for pagination

```
getProductListPaginate(thePage: number,
                       thePageSize: number,
                       theCategoryId: number): Observable<GetResponseProducts> {

  const url = `${this.baseUrl}/search/findByCategoryId`
    + `?id=${theCategoryId}&page=${thePage}&size=${thePageSize}`;

  return this.httpClient.get<GetResponseProducts>(url);
}
```

Spring Data REST supports pagination out of the box.
Just send the parameters for page and size

Step 4: Update ProductListComponent to handle pagination

[illegible]

[illegible]

[illegible] on right hand side of assignment is data from Spring Data REST JSON

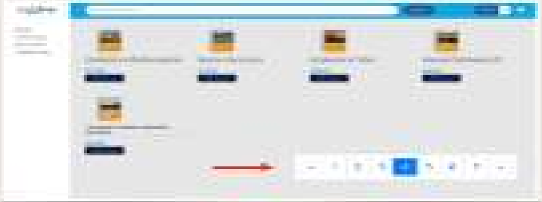
Step 4: Update ProductListComponent to handle pagination

[illegible]

Pagination component: pages are 1 based

Spring Data REST: pages are 0 based

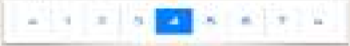
Step 4: Enhance HTML template to use ng-bootstrap pagination component



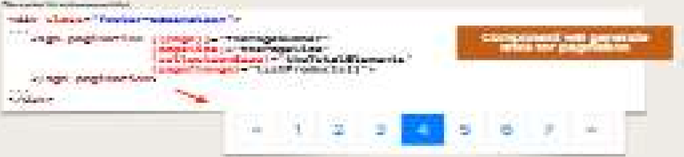
Step 3: Enhance HTML template to use ng-bootstrap pagination component

Parameter	Purpose
page	The page number to assume. 1-based … defaults to 1
pageSize	The size of the page (items per page). Defaults to 10
collectionSize	The total number of items
pageChange	Event handler for page change events

[illegible]



Step 3: Enhance HTML template to use ng-bootstrap pagination component



```
<div class="footer-pagination">
  ...
    <ngb-pagination [(page)]="managePageNumber"
                    [pageSize]="managePageSize"
                    [collectionSize]="listTotalElements"
                    (pageChange)="listProducts()">
    </ngb-pagination>
  ...
</div>
```

Component will generate links for pagination

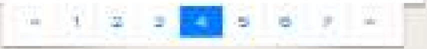
Step 3: Enhance HTML template to use ng-bootstrap pagination component

product-list-grid.component.html

```
<div class="footer-pagination">
...
    <ngb-pagination [(page)]="thePageNumber"

    </ngb-pagination>
</div>
```

Special Angular syntax for Two-Way Data Binding

Known as "Banana in a box"



Step 4: Enhance HTML template to use ng-bootstrap pagination component

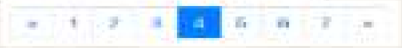
[illegible]

[illegible]

```
<div class="footer-pagination">
    <ngb-pagination [(page)]="[illegible]"
                    [illegible]

    </ngb-pagination>
</div>
```

[illegible]

```
export class [illegible] implements OnInit {

  [illegible]
  [illegible]
  [illegible]
  ...
}
```



Step 3: Enhance HTML template to use ng bootstrap pagination component

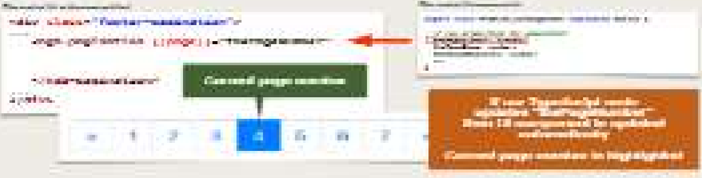
```
<div class="footer-pagination">
    <ngb-pagination [(page)]="[illegible]"
    
    </ngb-pagination>
</div>
```

When user clicks a "page" navigation link

In our TypeScript component "[illegible]" property is updated, based on user action

If user clicks 4, then "[illegible]" is set to 4

Step 3: Enhance HTML template to use ng-bootstrap pagination component



[illegible]

```
<div class="footer-pagination">
    <ngb-pagination [(page)]="currentpage"


    </ngb-pagination>
</div>
```

[illegible]

Current page number

« 1 2 3 4 5 6 7 »

If our TypeScript code updates "currentpage" then UI component is updated automatically

Current page number is highlighted

Step 3: Enhance HTML template to use ng-bootstrap pagination component

Special Angular syntax for One Way Data Binding

One-Way Data Binding

```
<ngb-pagination [(page)]="thePageNumber"
                [pageSize]="thePageSize"

</ngb-pagination>
</div>
```

```
[illegible]
```

[illegible]

Step 4: Enhance HTML template to use ng-bootstrap pagination component

two-way data binding

File: product-list.component.html

```
<div class="footer-pagination">
  ...
    <ngb-pagination [(page)]="thePageNumber"
                    [pageSize]="thePageSize"
                    [collectionSize]="theTotalElements">

    </ngb-pagination>
</div>
```

File: product-list.component.ts

```
export class ProductListComponent implements OnInit {

  // new properties for pagination
  thePageNumber: number = 1;
  thePageSize: number = 10;
  theTotalElements: number = 0;
}
```

If user clicks page link, updates "thePageNumber" field in component is updated automatically

Step 3: Enhance HTML template to use ng-bootstrap pagination component

File: product-list-grid.component.html

```
<div class="footer-pagination">
...
    <ngb-pagination [(page)]="thePageNumber"
                    [pageSize]="thePageSize"
                    [collectionSize]="theTotalElements"
                    (pageChange)="listProducts()">
    </ngb-pagination>
</div>
```

Event Binding:
pageChange event

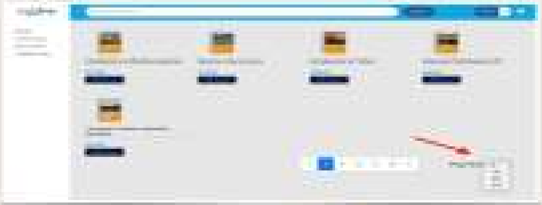
File: product-list.component.ts

```
export class ProductListComponent implements OnInit {
  listProducts() {

    this.searchMode = ...;

    if (this.searchMode) {
      this.handleSearchProducts();
    }
    else {
      this.handleListProducts();
    }
  }
}
```

When user clicks a "page" navigation link
then call the method
listProducts()

Pagination - Select Page Size

Pagination



Development Process

[illegible]

1. Add drop down list for page size in HTML template
2. Update ProductListComponent for setting page size

Step 1: Add drop-down list for page size to HTML template

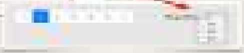
Event binding: on change

When user selects a value, then call method: updatePageSize(...)

The value that is selected by the user

5, 10, 20 or 50

```
<span class="mr-2">Page Size</span>

<select (change)="updatePageSize($event.target.value)">
    <option selected="true">5</option>
    <option>10</option>
    <option>20</option>
    <option>50</option>
</select>
```



Step 2: Update ProductListComponent for setting page size

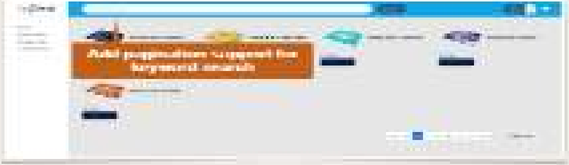
File: product-list.component.ts

```
export class ProductListComponent implements OnInit {

  // new properties for pagination
  thePageNumber: number = 1;
  thePageSize: number = 5;
  theTotalElements: number = 0;

  ...

  updatePageSize(pageSize: number) {
    this.thePageSize = pageSize;
    this.thePageNumber = 1;
    this.listProducts();
  }

}
```

Set page size based on parameter value

Pagination - Keyword Search

Pagination

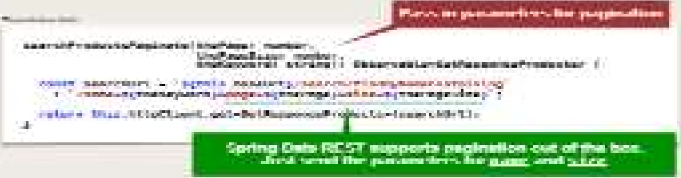


Add pagination support for keyword search

Development Process

1. Add pagination support to ProductService
2. Update ProductListComponent to handle pagination

Step1: Add pagination support to ProductService



product.service.ts

```
searchProductsPaginate(thePage: number,
                       thePageSize: number,
                       theKeyword: string): Observable<GetResponseProducts> {

  const searchUrl = `${this.baseUrl}/search/findByNameContaining`
    + `?name=${theKeyword}&page=${thePage}&size=${thePageSize}`;

  return this.httpClient.get<GetResponseProducts>(searchUrl);
}
```

Pass in parameters for pagination

Spring Data REST supports pagination out of the box.
Just send the parameters for page and size

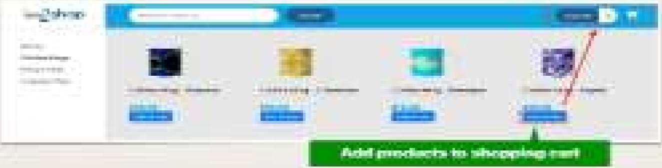
Step 2: Update ProductListComponent to handle pagination

```
[illegible]
```

[illegible]

Add Products to Shopping Cart - Part 1

Add Products to Shopping Cart



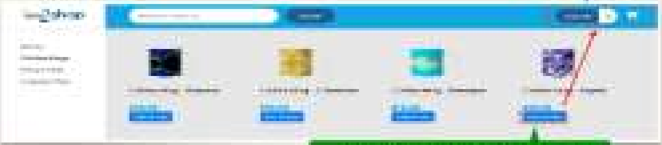
Add products to shopping cart

Overview of Entire Shopping Cart Process

1. Cart Status Component: on main page, display total price and quantity
2. Cart Details Page: list the items in the cart
3. Cart Details Page: add / remove items
4. Checkout Button
5. Checkout Form

We will break this up into multiple videos

Add Products to Shopping Cart



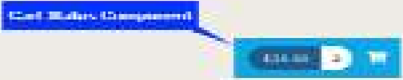
Get Sales Compared

Add products to shopping cart

Development Process - Part 1

1. Create new component: CartStatusComponent
2. Add HTML template for CartStatusComponent
3. Add click handler for "Add to cart" button
4. Update ProductListComponent with click handler method

Step 1: Create new component for Cart Status



```
> ng generate component components/cart-status
```

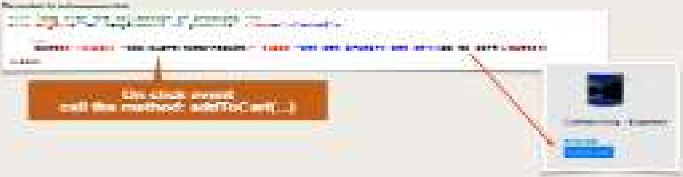
Step 2: Add HTML template for CartStatusComponent

File: cart-status.component.html

```
<div class="cart-area d-n">
    <a href="shopping-detail.html">
        <div class="total">$20.00
            <span>2</span>
        </div>
        <i class="fa fa-shopping-cart"></i>
    </a>
</div>
```

Font Awesome
Shopping Cart Icon

Step 3: Add click handler for "Add to cart" button



[illegible]

On-click event
call the method: addToCart(...)

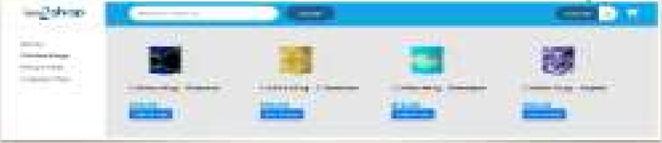
Step 4: Update ProductListComponent with click handler method

[illegible]

```
addToCart(theProduct: Product) {
  console.log(`Adding to cart: ${theProduct.name}, ${theProduct.unitPrice}`);
  // TODO ... do the real work
}
```

Add Products to Shopping Cart - Part 2

Add Products to Shopping Cart



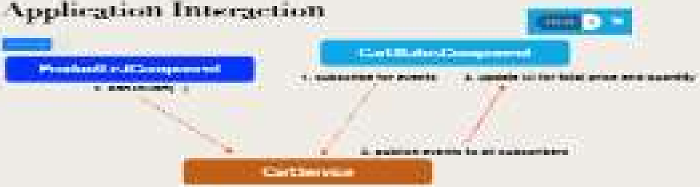
Update status

Development Process - Part 2

[illegible]

1. Create model class: CartItem
2. Develop CartService
3. Modify ProductListComponent to call CartService
4. Enhance CartStatusComponent to subscribe to CartService
5. Update CartStatusComponent HTML to display cart total price and quantity

Application Interaction



ProductListComponent

CartStatusComponent

1. subscribe for events

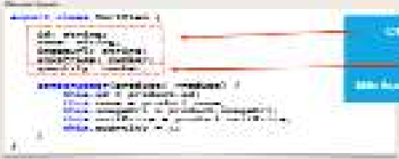
2. addToCart()

3. publish events to all subscribers

4. update UI for total price and quantity

CartService

Step 1: Create model class: CartItem



File: cart-item.ts

```
export class CartItem {

    id: string;
    name: string;
    imageUrl: string;
    unitPrice: number;
    quantity: number;

    constructor(product: Product) {
        this.id = product.id;
        this.name = product.name;
        this.imageUrl = product.imageUrl;
        this.unitPrice = product.unitPrice;
        this.quantity = 1;
    }
}
```

Contains essential fields of Product
for use in the cart

Also has a field for quantity

We have "a Variation of "Product"" in our app

Step 2: Develop CartService

```
@Injectable({
export class CartService {

  cartItems: CartItem[] = [];

  totalPrice: Subject<number> = new Subject<number>();
  totalQuantity: Subject<number> = new Subject<number>();

  ...
}
```

Subject is a subclass of Observable

[illegible]

[illegible]

Step 2: Develop CartService

[illegible]

Step 2: Develop CartService

[illegible]

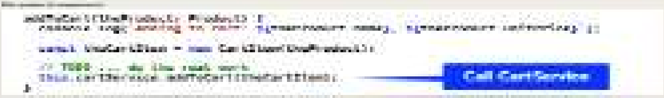
```
[illegible]
```

Compile fails!

This will publish events to all subscribers

[illegible]

Step 3: Modify ProductListComponent to call CartService



```
addToCart(theProduct: Product) {
  console.log(`Adding to cart: ${theProduct.name}, ${theProduct.unitPrice}`);

  const theCartItem = new CartItem(theProduct);

  // TODO ... do the real work
  this.cartService.addToCart(theCartItem);
}
```

Call CartService

Step 4: Enhance CartStatusComponent to subscribe to CartService

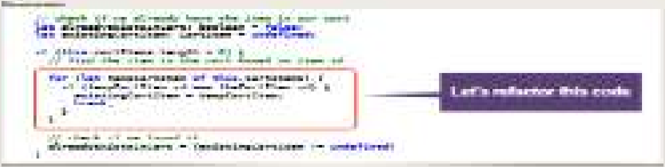
When new results are observed, make the assignments to update UI

Step 8: Update CartStatusComponent HTML to display cart total price and quantity

```
[illegible]
```

Refactor Cart Service

Currently: Basic code for finding item in cart

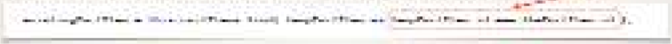


```
// check if we already have the item in our cart
let alreadyExistsInCart: boolean = false;
let existingCartItem: CartItem = undefined;

if (this.cartItems.length > 0) {
  // find the item in the cart based on item id

  for (let tempCartItem of this.cartItems) {
    if (tempCartItem.id === theCartItem.id) {
      existingCartItem = tempCartItem;
      break;
    }
  }

  // check if we found it
  alreadyExistsInCart = (existingCartItem != undefined);
}
```

Let's refactor this code

Array.find(...)

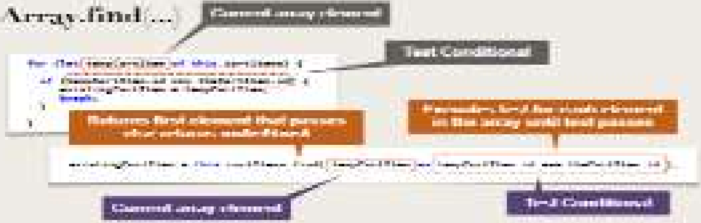
- Method returns the first element from array that passes a given test



```
[illegible]
```

- Executes the test for each element in the array until the test passes
- If test passes, then returns the first element in the array that passed
- If test fails for ALL elements in the array, then returns undefined

Array.find(...)



```
for (let tempProfile of this.profiles) {
  if (tempProfile.id === [illegible]) {
    existingProfile = tempProfile;
    break;
  }
}
```

```
existingProfile = this.profiles.find(tempProfile => tempProfile.id === [illegible]);
```

Current array element
Test Conditional
Returns first element that passes otherwise, undefined
Executes test for each element in the array until test passes
Current array element
Test Conditional

Refactored: Before and After

BEFORE

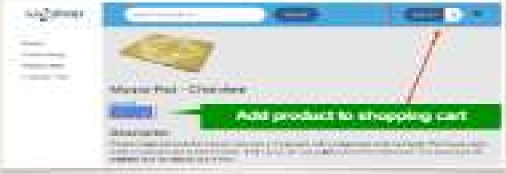
```
[illegible]
```

AFTER

```
[illegible]
```

Add Products to Cart - Details View

Add Products to Cart - Details View



Add product to shopping cart

Development Process

1. Add click handler for "Add to cart" button on product-details.component.html

2. Update ProductDetailsComponent with click handler method

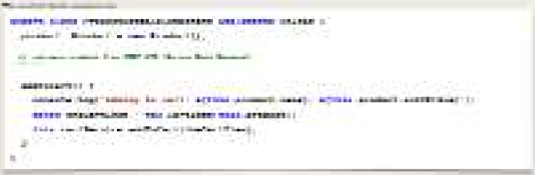
Step 1: Add click handler for "Add to cart" button

[illegible]

```
<button (click)="addToCart()" class="btn btn-primary btn-sm">Add to cart</button>
```

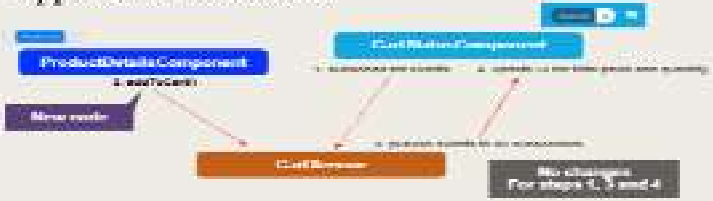
On click event
call the method addToCart()

Step 2: Update ProductDetailsComponent with click handler method



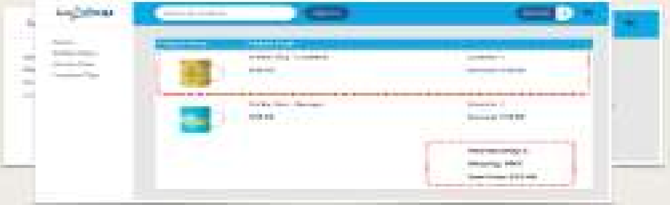
```
export class ProductDetailsComponent implements OnInit {
  product!: Product = new Product();

  // [illegible]

  addToCart() {
    console.log(`adding to cart: ${this.product.name}, ${this.product.unitPrice}`);
    const theCartItem = new CartItem(this.product);
    this.cartService.addToCart(theCartItem);
  }
}
```

Application Interaction

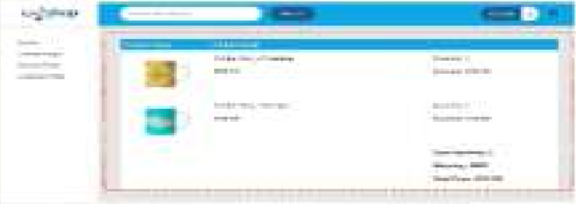


Shopping Cart - List Items

List Items in the Shopping Cart



New Component: CartDetailsComponent

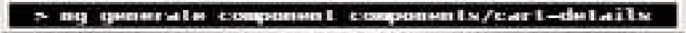


Development Process

Step-By-Step

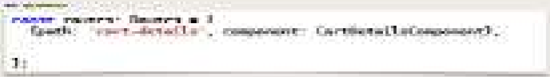
1. Create new component: CartDetailsComponent
2. Add new route for CartDetailsComponent
3. Update link for Shopping Cart icon
4. **Modify CartDetailsComponent to retrieve cart items**
5. Add HTML template for CartDetailsComponent

Step 1: Create new component for Cart Details



```
> ng generate component components/cart-details
```

Step 2: Add new route for CartDetailsComponent



[illegible]

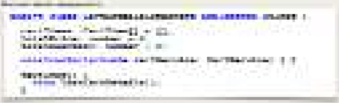
```
const routes: Routes = [
  {path: 'cart-details', component: CartDetailsComponent},

];
```

Step 3: Update link for Shopping Cart icon

```
<div class="cart-area d-n">

    <a routerLink="/cart-details">
        <div class="total">{{ totalPrice | currency: 'USD' }}
            <span>{{ totalQuantity }}</span>
        </div>
        <i class="fa fa-shopping-cart" aria-hidden="true"></i>
    </a>

</div>
```

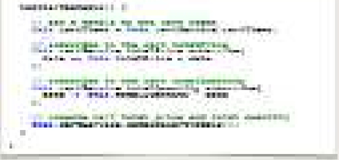
When user clicks cart status component
then we route to /cart-details
to show CartDetailsComponent

Step 4: Modify CartDetailsComponent to retrieve cart items



File: cart-details.component.ts

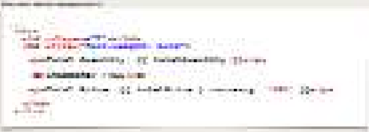
```
export class CartDetailsComponent implements OnInit {

  cartItems: CartItem[] = [];
  totalPrice: number = 0;
  totalQuantity: number = 0;

  constructor(private cartService: CartService) { }

  ngOnInit(): void {
    this.listCartDetails();
  }
```



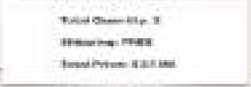
```
  listCartDetails() {

    // get a handle to the cart items
    this.cartItems = this.cartService.cartItems;

    // subscribe to the cart totalPrice
    this.cartService.totalPrice.subscribe(
      data => this.totalPrice = data
    );

    // subscribe to the cart totalQuantity
    this.cartService.totalQuantity.subscribe(
      data => this.totalQuantity = data
    );

    // compute cart total price and quantity
    this.cartService.computeCartTotals();
  }

}
```

Step 5: Add HTML template for CartDetailsComponent

Step 5: Add HTML template for CartDetailsComponent



```
[illegible]
```



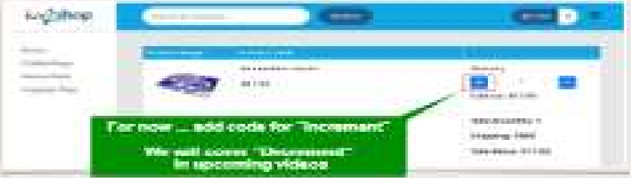
Total Quantity: 3

Shipping: FREE

Total Price: $217.00

Shopping Cart - Increment Item Quantity

Increment Item Quantity in the Shopping Cart



For now … add code for "Increment"

We will cover "Decrement" in upcoming videos

Development Process

1. Modify CartDetailsComponent HTML template
   1. Add the "Increment" button
   2. Add click handler for the "Increment" button on HTML template

2. Update CartDetailsComponent with click handler method

Step 1: Modify CartDetailsComponent HTML template

[illegible]

```
[illegible]
```

On click event
call the method: incrementQuantity(...)

Font Awesome
"Plus" icon

Step 2: Update CartDetailsComponent with click handler method

File: cart-details.component.ts

```
incrementQuantity(theCartItem: CartItem) {
  this.cartService.addToCart(theCartItem);
}
```

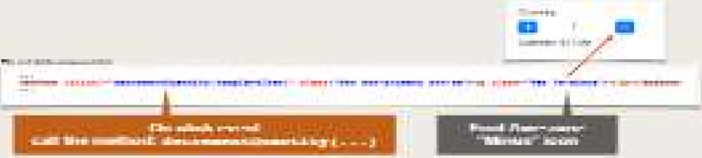
Shopping Cart - Decrement / Remove Item

Decrement / Remove Item from the Shopping Cart

Development Process

1. Add click handler for the "decrement" button on HTML template
2. Update CartDetailsComponent with click handler method
3. Modify CartService with supporting method
4. Repeat process for "Remove" button

Step 1: Modify CartDetailsComponent HTML template



On click event
call the method: decrementQuantity(...)

Font Awesome
"Minus" icon

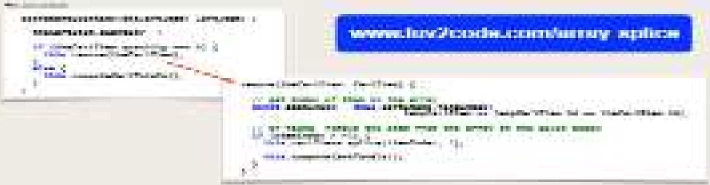
Step 2: Update CartDetailsComponent with click handler method

cart-details.component.ts

```
decrementQuantity(theCartItem: CartItem) {
  this.cartService.decrementQuantity(theCartItem);
}
```

Step 3: Modify CartService with supporting method

www.luv2code.com/array-splice



```
decrementQuantity(theCartItem: CartItem) {
  theCartItem.quantity--;

  if (theCartItem.quantity === 0) {
    this.remove(theCartItem);
  }
  else {
    this.computeCartTotals();
  }
}
```

```
remove(theCartItem: CartItem) {

  // get index of item in the array
  const itemIndex = this.cartItems.findIndex(
                       tempCartItem => tempCartItem.id === theCartItem.id );

  // if found, remove the item from the array at the given index
  if (itemIndex > -1) {
    this.cartItems.splice(itemIndex, 1);

    this.computeCartTotals();
  }
}
```

Step 4: "Remove" button

[illegible]

```
[illegible]
```

On click event
call the method remove(...)

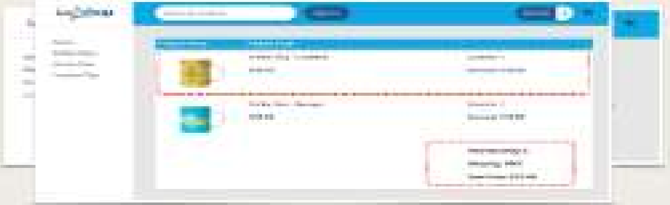
Step 4: "Remove" button

[illegible]

```
removeFromCart(item: CartItem) {
  this.cartService.removeFromCart(item);
}
```

Shopping Cart - List Items

List Items in the Shopping Cart



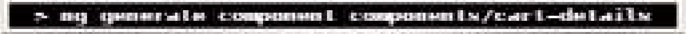
New Component: CartDetailsComponent

Development Process

Step-By-Step

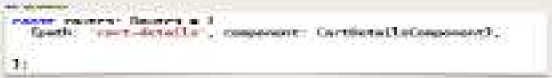
1. Create new component: CartDetailsComponent
2. Add new route for CartDetailsComponent
3. Update link for Shopping Cart icon
4. Modify CartDetailsComponent to retrieve cart items
5. Add HTML template for CartDetailsComponent

Step 1: Create new component for Cart Details



```
> ng generate component components/cart-details
```

Step 2: Add new route for CartDetailsComponent



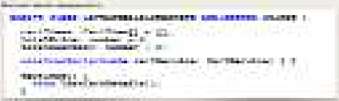
```
const routes: Routes = [
  {path: 'cart-details', component: CartDetailsComponent},

];
```

Step 3: Update link for Shopping Cart icon

```
<div class="cart-area d-n">

    <a routerLink="/cart-details">
        <div class="total">{{ totalPrice | currency: 'USD' }}
            <span>{{ totalQuantity }}</span>
        </div>
        <i class="fa fa-shopping-cart" aria-hidden="true"></i>
    </a>

</div>
```

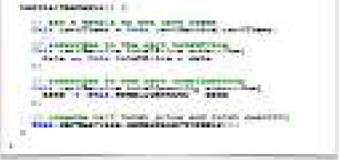
When user clicks cart status component
then use new route /cart-details
to show CartDetailsComponent

Step 4: Modify CartDetailsComponent to retrieve cart items



cart-details.component.ts

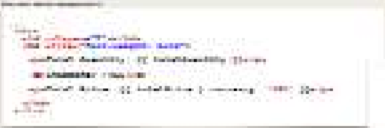
```
export class CartDetailsComponent implements OnInit {

  cartItems: CartItem[] = [];
  totalPrice: number = 0;
  totalQuantity: number = 0;

  constructor(private cartService: CartService) { }

  ngOnInit() {
    this.listCartDetails();
  }
```



```
  listCartDetails() {

    // get a handle to the cart items
    this.cartItems = this.cartService.cartItems;

    // subscribe to the cart totalPrice
    this.cartService.totalPrice.subscribe(
      data => this.totalPrice = data
    );

    // subscribe to the cart totalQuantity
    this.cartService.totalQuantity.subscribe(
      data => this.totalQuantity = data
    );

    // compute cart total price and total quantity
    this.cartService.computeCartTotals();
  }

}
```

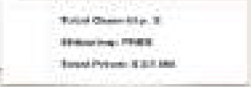
Step 5: Add HTML template for CartDetailsComponent

Step 5: Add HTML template for CartDetailsComponent



[illegible]

```
[illegible]
```



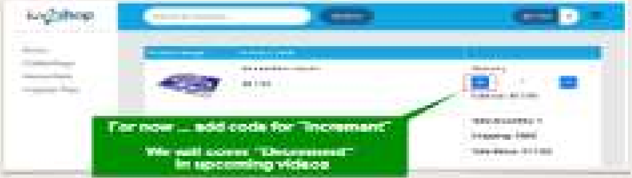
Total Quantity: 3

Shipping: FREE

Total Price: [illegible]

Shopping Cart - Increment Item Quantity

Increment Item Quantity in the Shopping Cart



For now ... add code for "Increment"

We will cover "Decrement" in upcoming videos

Development Process

1. Modify CartDetailsComponent HTML template
    1. Add the "Increment" button
    2. Add click handler for the "Increment" button on HTML template
2. Update CartDetailsComponent with click handler method

Step 1: Modify CartDetailsComponent HTML template

[illegible]

```
[illegible]
```

On click event
call the method: [illegible](...)

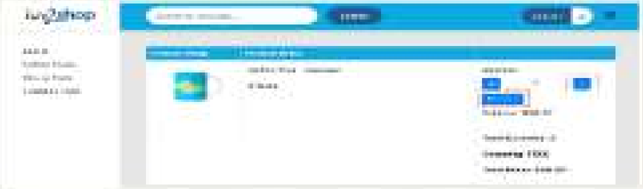
Font Awesome
"Plus" icon

Step 2: Update CartDetailsComponent with click handler method

```
incrementQuantity(theCartItem: CartItem) {
  this.cartService.addToCart(theCartItem);
}
```

Shopping Cart - Decrement / Remove Item

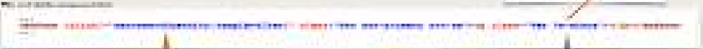
Decrement / Remove Item from the Shopping Cart



Development Process

1. Add click handler for the "decrement" button on HTML template
2. Update CartDetailsComponent with click handler method
3. Modify CartService with supporting method
4. Repeat process for "Remove" button

Step 1: Modify CartDetailsComponent HTML template

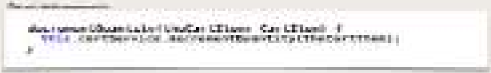


```
[illegible]
```

On click event
call the method: decrementQuantity(...)

Font Awesome
"Minus" icon

Step 2: Update CartDetailsComponent with click handler method

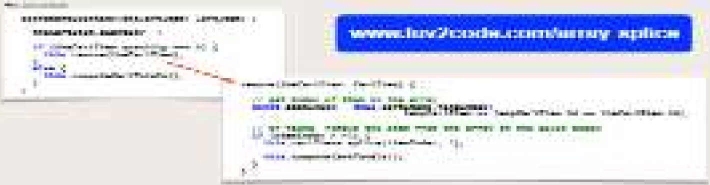


File: cart-details.component.ts

```
decrementQuantity(theCartItem: CartItem) {
  this.cartService.decrementQuantity(theCartItem);
}
```

Step 3: Modify CartService with supporting method

www.luv2code.com/array-splice



```
decrementQuantity(theCartItem: CartItem) {
  theCartItem.quantity--;

  if (theCartItem.quantity === 0) {
    this.remove(theCartItem);
  }
  else {
    this.computeCartTotals();
  }
}
```

```
remove(theCartItem: CartItem) {

  // get index of item in the array
  const itemIndex = this.cartItems.findIndex(
                          tempCartItem => tempCartItem.id === theCartItem.id);

  // if found, remove the item from the array at the given index
  if (itemIndex > -1) {
    this.cartItems.splice(itemIndex, 1);

    this.computeCartTotals();
  }
}
```

Step 4: "Remove" button

[illegible]

```
[illegible]
```

On click event
call the method: remove(...)

Step 4: "Remove" button

[illegible]

```
removeItem(cartItem: CartItem) {
  this.cartService.removeItem(cartItem);
}
```

Checkout Form - Layout

Overview of Angular Forms

- Can easily build forms in Angular
- Supports form data-binding, validation and processing
- Angular provides two types of forms
  - Reactive forms
  - Template-driven forms

Forms

- Reactive forms
  - Leverages programmable API for form building
  - Scalable solution that is designed for large, complex forms
  - Forms can be easily created and tested

- Template-driven forms
  - Designed for small, simple forms
  - Not a scalable solution for large, complex forms

We will use Reactive forms.

Comparison

- For comparison of Reactive forms and Template driven forms.

https://angular.io/guide/forms-overview

Key Components

Name	Description
FormControl	Individual control that tracks the value and validation status
FormGroup	A collection of controls. Can contain nested groups.
others	

Our Checkout Form

Focus on form usability and layout

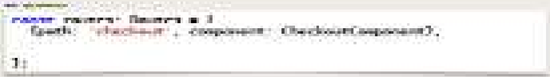
Development Process

1. Generate new checkout component
2. Add a new route for checkout component
3. Create a new checkout button and link to checkout component
4. Add support for reactive forms
5. Define form in component .ts file
6. Layout form controls in HTML template
7. Add event handler for form submission

Step 1: Generate our checkout component

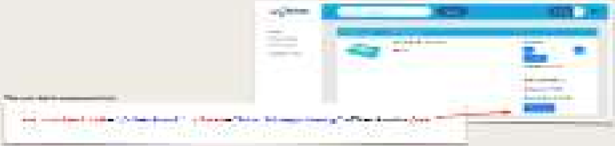
```
> ng generate component components/checkout
```

Step 2: Add a new route for checkout component



```
[illegible]
const routes: Routes = [
  {path: 'checkout', component: CheckoutComponent},

];
```

Step 3: Create new checkout button and link to checkout component



[illegible]

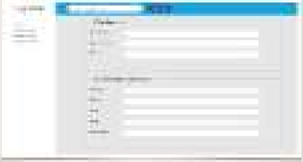
```
[illegible]
```

Step 4: Add support for reactive forms

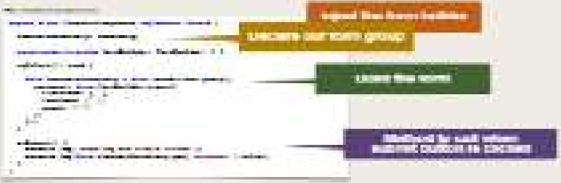
[illegible]

```
imports: [
  ...
  ReactiveFormsModule
],
```

Step 5: Define form in component .ts file



Step 5: Define form in component .ts file



```
[illegible]
```

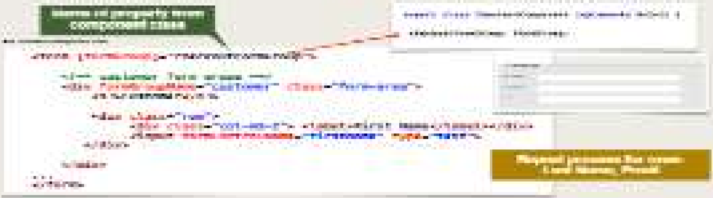
Inject the form builder

Declare our form group

[illegible] the form

Method to call when submit button is clicked

Step 6: Layout form controls in HTML template



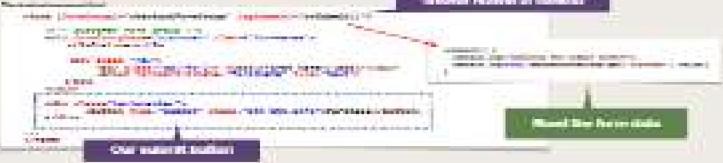
```
export class CheckoutComponent implements OnInit {

  checkoutFormGroup: FormGroup;
```

```
<form [formGroup]="checkoutFormGroup">

    <!-- customer form group -->
    <div formGroupName="customer" class="form-area">
        <h3>Customer</h3>

        <div class="row">
            <div class="col-md-2"> <label>First Name</label></div>
            <input formControlName="firstName" type="text">
        </div>

    </div>
...
</form>
```

Name of property from component class

Repeat process for Last Name, Email

Step 7: Add event handler for form submission



Method to call when submit button is clicked

Read the form data

Our submit button

Step 7: Add event handler for form submission

[illegible]

Output

[illegible]

Our Checkout Form

Repeat the process for other sections of the form

Checkout Form
Populate Credit Card Expiration Dates

Credit Card Expiration Dates

Populating Drop-Down Lists

- We could just hard code the values ... but not ideal :-(
- Years
  - Years will become outdated ... if we are in year 2022, why show 2020???
  - Dynamically populate current year ... up to next 10 years

Development Process

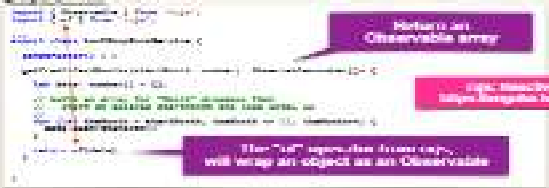
Step-By-Step

1. Create our form service Luv2ShopFormService.ts
2. Add methods to the form service for months and years
3. Update checkout component to retrieve the months and years from service
4. Update HTML template to populate drop down lists for months and years

Step 1: Generate our form service

```
$ ng generate service services/Luv2ShopForm
```

Step 2: Add methods to form service for months and years



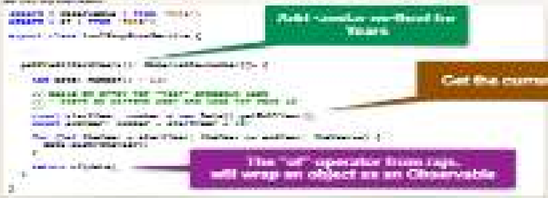
```
luv2-shop-form.service.ts
import { Observable } from 'rxjs';
import { of } from 'rxjs';

@Injectable()
export class Luv2ShopFormService {

  constructor() { }

  getCreditCardMonths(startMonth: number): Observable<number[]> {

    let data: number[] = [];

    // build an array for "Month" dropdown list
    //   - start at current month and loop until end of year

    for (let theMonth = startMonth; theMonth <= 12; theMonth++) {
      data.push(theMonth);
    }

    return of(data);
  }

}
```

Return an Observable array

rxjs: Reactive JavaScript
https://angular.io/guide/rx-library

The "of" operator from rxjs, will wrap an object as an Observable

Step 2: Add methods to form service for months and years



```
import { Observable } from 'rxjs';
import { of } from 'rxjs';

export class Luv2ShopFormService {

  getCreditCardYears(): Observable<number[]> {

    let data: number[] = [];

    // build an array for "Year" downlist list
    // - start at current year and loop for next 10 years

    const startYear: number = new Date().getFullYear();
    const endYear: number = startYear + 10;

    for (let theYear = startYear; theYear <= endYear; theYear++) {
      data.push(theYear);
    }

    return of(data);
  }
}
```

Add similar method for Years

Get the current year

The "of" operator from rxjs, will wrap an object as an Observable

Step 3: Update checkout component to retrieve months and years

```
export class CheckoutComponent implements OnInit {

  ...
  creditCardYears: number[] = [];
  creditCardMonths: number[] = [];

  constructor(private formBuilder: FormBuilder,
              private luv2ShopFormService: Luv2ShopFormService) { }

  ngOnInit(): void {

    // populate credit card months
    const startMonth: number = new Date().getMonth() + 1;
    console.log("startMonth: " + startMonth);

    this.luv2ShopFormService.getCreditCardMonths(startMonth).subscribe(
      [illegible]
    );
  }
}
```

Inject our form service

Get the current month

JavaScript Date object: the months are zero-based (0 .. 11)

So we add 1 to get (1 .. 12)

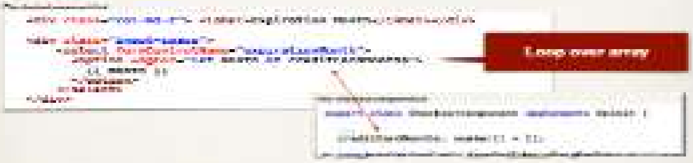
www.luv2code.com/javascript-date

Step 3: Update checkout component to retrieve months and years

```
export class CheckoutComponent implements OnInit {

  creditCardMonths: number[] = [];
  creditCardYears: number[] = [];

  constructor(private formBuilder: FormBuilder,
              private luv2ShopFormService: Luv2ShopFormService) { }

  ngOnInit(): void {

    // populate credit card years
    this.luv2ShopFormService.getCreditCardYears().subscribe(
      data => {
        console.log("Retrieved credit card years: " + JSON.stringify(data));
        this.creditCardYears = data;
      }
    );
  }
}
```

Get the
credit card years

Step 4: Update HTML template to populate drop down lists



checkout.component.html

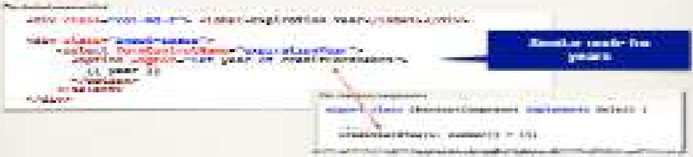
```
<div class="col-md-2"> <label>Expiration Month</label></div>

<div class="input-space">
    <select formControlName="expirationMonth">
        <option *ngFor="let month of creditCardMonths">
            {{ month }}
        </option>
    </select>
</div>
```

Loop over array

checkout.component.ts

```
export class CheckoutComponent implements OnInit {

  creditCardMonths: number[] = [];
```

Step 4: Update HTML template to populate drop down lists



checkout.component.html

```
<div class="col-md-2"> <label>Expiration Year</label></div>

<div class="input-space">
    <select formControlName="expirationYear">
        <option *ngFor="let year of creditCardYears">
            {{ year }}
        </option>
    </select>
</div>
```

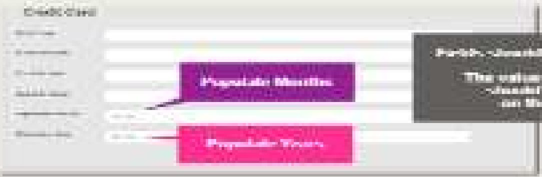
Similar code for years

checkout.component.ts

```
export class CheckoutComponent implements OnInit {
  ...
  creditCardYears: number[] = [];
```

Checkout Form Dependent Fields

Dependent Fields



Populate Months

Populate Years

Fields should be dependent

The values of month should depend on the year

Dependent Fields

- Values for month should depend on the year

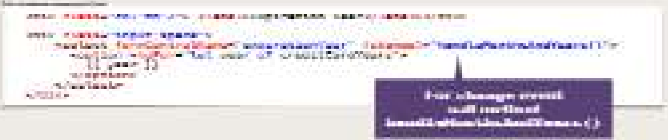
- If the current year is selected
  - Then only show the remaining months for the year
  - Start at current month to 12

- If a future year is selected
  - Then show months: 1 - 12

Development Process

1. Update HTML template
    1. For the Expiration years drop-down list
    2. Add event binding for change event
2. Update checkout component, add event handler
    1. Read the selected year
    2. Update the list of months based on selected year

Step 1: Update HTML template for event binding



```
<div class="col-md-2"> <label class="col-form-label">Expiration Year</label></div>

<div class="col-md-9 input-space">
    <select formControlName="expirationYear" (change)="handleMonthsAndYears()">
        <option *ngFor="let year of creditCardYears">
            {{ year }}
        </option>
    </select>
</div>
```

For change event call method handleMonthsAndYears()

Step 2: Update checkout component, add event handler

```
[illegible]
```

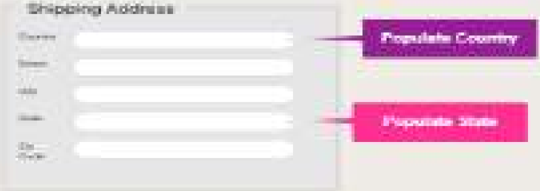
Read the selected year from the form

Get the current month

Get the credit card months

Checkout Form
Populate Country-State Drop-Down Lists

Address - Country and State



Populate Country

Populate State

Populating Drop-Down Lists

- Populate countries and states from the backend REST API (database)
- The user will select a country
- Depending on country selected, populate list of states
  - Term "State" is different in each country
    - Similar to "Province", "District", "Region", "Prefecture" etc

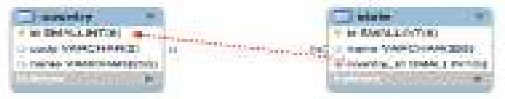
Development Process - Backend

1. Create database tables
2. Develop JPA Entities (Country, State)
3. Create Spring Data Repositories
4. Update Spring Data REST Config

Step 1: Create Database Tables

One country has many states

Many states belong to one country

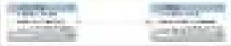


Step 1: Create Database Tables

```
CREATE TABLE country (
  id smallint unsigned NOT NULL,
  code varchar(2) UNIQUE NULL,
  name varchar(100) UNIQUE NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB;
```

```
CREATE TABLE town (
  id smallint unsigned NOT NULL AUTO_INCREMENT,
  name varchar(100) UNIQUE NULL,
  country_id smallint unsigned NOT NULL,
  PRIMARY KEY (id),
  KEY fk_country (country_id),
  CONSTRAINT fk_country FOREIGN KEY (country_id) REFERENCES country (id)
) ENGINE=InnoDB AUTO_INCREMENT=1;
```

Step 1: Create Database Tables



```
INSERT INTO `country` VALUES
(1, 'BR', 'Brazil'),
(2, 'FR', 'France'),
(3, 'DE', 'Germany'),
(4, 'IN', 'India'),
(5, 'TR', 'Turkey'),
(6, 'US', 'United States');
```

id, code, name

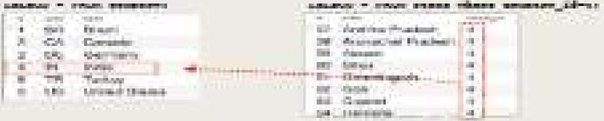
```
INSERT INTO `brand` VALUES
[illegible]
[illegible]
[illegible]
...
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
```

Brazil: 1

India: 4

id, name, country

Step 1: Create Database Tables



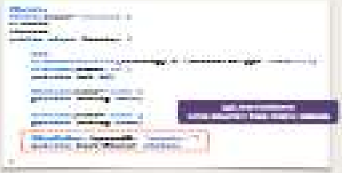
SELECT * FROM COUNTRY;

id	code	name
1	BR	Brazil
2	CA	Canada
3	DE	Germany
4	IN	India
5	TR	Turkey
6	US	United States

SELECT * FROM STATE WHERE COUNTRY_ID=4;

id	name	country_id
37	Andhra Pradesh	4
38	Arunachal Pradesh	4
39	Assam	4
40	Bihar	4
41	Chhattisgarh	4
42	Goa	4
43	Gujarat	4
44	Haryana	4

Step 2: Develop JPA Entities: Country and State



[illegible]

[illegible]

Step 3: Create Spring Data REST Repositories

```
@CrossOrigin("http://localhost:4200")
@RepositoryRestResource(collectionResourceRel = "countries", path = "countries")
public interface CountryRepository extends JpaRepository<Country, Integer> {
}
```

Expose /countries endpoint

To retrieve all countries

http://localhost:8080/api/countries

To retrieve country id 4

http://localhost:8080/api/countries/4

Step 3: Create Spring Data REST Repositories

To retrieve all countries

http://localhost:8080/api/countries

To retrieve country id 1

http://localhost:8080/api/countries/1

Step 3: Create Spring Data REST Repositories

```
@CrossOrigin("http://localhost:4200")
@RepositoryRestResource
public interface StateRepository extends JpaRepository<State, Integer> {

}
```

Expose /states endpoint

To retrieve all states

http://localhost:8080/api/states

Step 3: Create Spring Data REST Repositories

```
@CrossOrigin("http://localhost:4200")
@RepositoryRestResource
public interface StateRepository extends JpaRepository<State, Integer> {

    List<State> findByCountryCode(@Param("code") String code);
}
```

To retrieve states for a given country code

http://localhost:8080/api/states/search/findByCountryCode?code=IN

http://localhost:8080/api/states/search/findByCountryCode?code=BR

Query param
Default Spring path

Step 3: Create Spring Data REST Repositories

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Step 4: Update Spring Data REST Configs

- Update our config for MyDataRestConfig.java

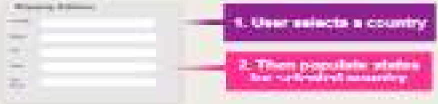
- Make the APIs for /country and /state read-only
  - This is reference data, no need to change it via REST API
  - **Disable HTTP: PUT, POST, etc …**

- Coding details covered in next slides

Checkout Form
Populate Country-State Drop-Down Lists

Populating Drop-Down Lists

- Backend work for REST API is done ... now let's handle front-end
- The user will select a country
- Depending on country selected, populate list of states



1. User selects a country

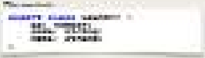
2. Then populate states for selected country

Development Process - Frontend

1. Create TypeScript classes for Country and State
2. Add methods to the form service for countries and states
3. Update checkout component to retrieve the countries from service
4. Update HTML template to populate drop-down lists for countries
5. Add event handler for checkout component
   1. Read the selected country; retrieve list of states based on selected country
6. Update HTML template to populate drop-down lists for states

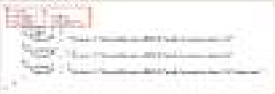
Backend REST APIs

Step 1: Create TypeScript classes for: Country and State



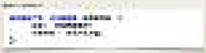
[illegible]

```
export class Country {
    id: number;
    code: string;
    name: string;
}
```



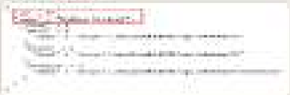
JSON from REST API

```
[illegible]
```



[illegible]

```
export class State {
    id: number;
    name: string;
}
```



JSON from REST API

```
[illegible]
```

Step 2: Add methods to the form service for countries and states

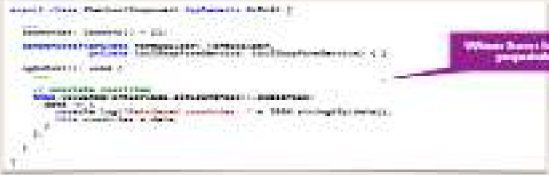
```
[illegible]
export class [illegible] {

  [illegible]
  [illegible]

  constructor([illegible]) { }

  [illegible] {

    // [illegible]
    [illegible]

    return this.[illegible](
      [illegible]
    );
  }

}

[illegible] {
  [illegible]: {
    [illegible]
  }
}
```

[illegible]

[illegible]

[illegible]

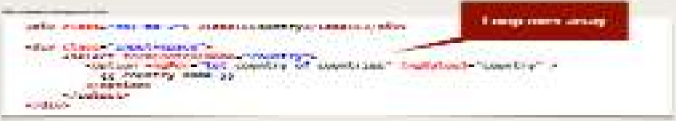
Step 3: Update checkout component to retrieve countries from service



```
export class CheckoutComponent implements OnInit {
  ...
  countries: Country[] = [];

  constructor(private formBuilder: FormBuilder,
              private luv2ShopFormService: Luv2ShopFormService) { }

  ngOnInit(): void {
    ...

    // populate countries
    this.luv2ShopFormService.getCountries().subscribe(
      data => {
        console.log("Retrieved countries: " + JSON.stringify(data));
        this.countries = data;
      }
    );
  }
}
```

When form is initially displayed, populate the countries

Step 4: Update HTML template to populate drop-down lists for countries



```
<div class="col-md-2"> [illegible]</div>

<div class="input-space">
    <select formControlName="country">
        <option *ngFor="let country of countries" [ngValue]="country" >
            {{ country.name }}
        </option>
    </select>
</div>
```

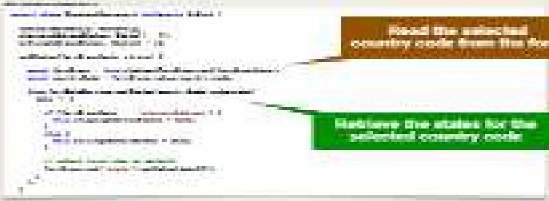
Loop over array

Step 5: Update checkout component, add event handler

For change event call method getStates(...)

```
<div class="col-md-2"> <label>Country</label></div>

<div class="input-space">
    <select formControlName="country" (change)="getStates('shippingAddress')">
        <option *ngFor="let country of countries" [ngValue]="country">
            {{ country.name }}
        </option>
    </select>
</div>
```

Step 5: Update checkout component, add event handler



```
[illegible]
```

Read the selected country code from the form

Retrieve the states for the selected country code

Step 6: Update HTML template to populate drop-down lists for states

```
<div class="col-md-2"> <label>State</label></div>

<div class="input-space">
    <select formControlName="state">
        <option *ngFor="let state of shippingAddressStates" [ngValue]="state">
            {{ state.name }}
        </option>
    </select>
</div>
```

Loop over array

Checkout Form Validation

Form Validation

- Before we submit the form to the backend … let's perform validation

- Check for required fields, min length etc …

Angular Validation

- Angular has a set of built-in validation rules.

Name	Description
required	Must be a non-empty value
min	Must be a number >= value
max	Must be a number <= value
minLength	Length must be >= value
maxLength	Length must be <= value
pattern	Must match a regular expression pattern
email	Validate email against common regular expression patterns
others ...	

https://angular.io/api/forms/Validators

Additional Validation Features

- Define custom validators
- Cross-field validation
- Asynchronous validators

https://angular.io/guide/form-validation

Checkout Form Validation

From: The Boss

- Our requirements:
  - All fields are required
  - Email address: has proper email format
  - Credit card field: only numbers allowed (16 digits)
  - CVC number: only numbers allowed (3 digits)

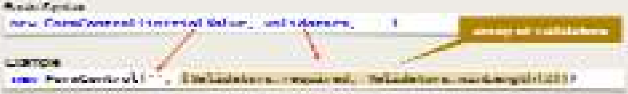
Development Process

[illegible]

1. Specify validation rules for the form controls
2. Define Getter methods to access form controls
3. Update HTML template to display error messages
4. Add event handler to check validation status when submit button clicked

Step 1: Specify validation rules for form controls

- Recall that a form field is represented by FormControl object
- Create the FormControl and pass in initial value and validators



Basic Syntax

```
new FormControl(initialValue, validators,     )
```

Example

```
new FormControl('', [Validators.required, Validators.minLength(2)])
```

array of validators

Step 1: Specify validation rules for form controls

```
[illegible]
```

Self/Cross form control

RegEx Expression

Validators.email vs Validators.pattern

- You may wonder why we aren't using Angular Validators.email ???
- Validators.email only checks for <some text>@<some text>
  - Based on this ... the following is valid email format:
    - myemail@luv2code
    - myemail@gmail

That's awkward :-(

- It does not check for domain extension (e.g. myemail@luv2code.com or myemail@gmail.com)
- As a result, we use a regular expression with Validators.pattern

Validators.email vs Validators.pattern

- You may wonder why we aren't using Angular's Validators.email ???
- Validators.email [illegible]
  - Basic [illegible]
    - [illegible]
    - [illegible]

Validators.email and Validators.pattern

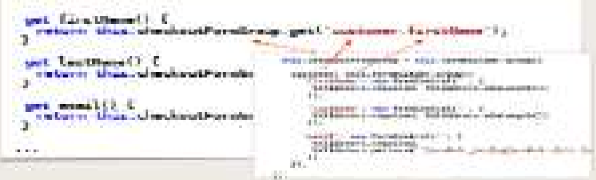
Only check the FORMAT

Does not verify if email address is real

It does not check for domain name (a TLD), myself@localhost or myself@gmail.com

As a result, we use a regular expression with Validators.pattern

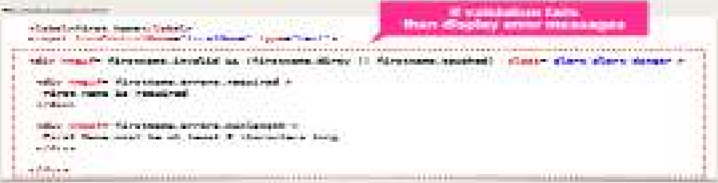
Step 2: Define Getter methods to access form controls



```
get firstName() {
  return this.checkoutFormGroup.get('customer.firstName');
}

get lastName() {
  return this.checkoutFormGr

get email() {
  return this.checkoutFormGr
}

...
```

```
this.checkoutFormGroup = this.formBuilder.group({
  customer: this.formBuilder.group({
    firstName: new FormControl('', [
      Validators.required, Validators.minLength(2)]
    ]),

    lastName: new FormControl('', [
      Validators.required, Validators.minLength(2)])
    ]),

    email: new FormControl('', [
      Validators.required,
      Validators.pattern('^[a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,4}$')]
    ]),
  }),
});
```

Step 3: Update HTML template to display error messages



If validation fails
then display error messages

```
<label>First Name</label>
<input formControlName="firstName" type="text">

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required">
    First Name is required
  </div>

  <div *ngIf="firstName.errors.minlength">
    First Name must be at least 2 characters long
  </div>

</div>
```

Step 3: Update HTML template to display error messages

checkout.component.html

invalid: did validation fail?

```
<label>First Name</label>
<input formControlName="firstName" type="text">

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required">
    First Name is required
  </div>

  <div *ngIf="firstName.errors.minlength">
    First Name must be at least 2 characters long
  </div>

</div>
```

Step 3: Update HTML template to display error messages

"dirty": did user change field value?

```
<label>First Name</label>
<input formControlName="firstName" id="firstName" type="text"/>

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

</div>
```

Only display validation errors if user has interacted with the form

- When the user changes field value, the control is marked as "dirty"
- When the field loses focus, the control is marked as "touched"

Step 3: Update HTML template to display error messages

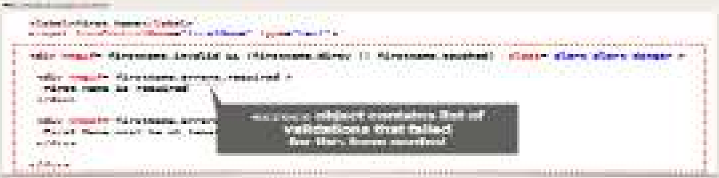
[illegible]

touched: did field lose focus?

```
<label>First Name</label>
<input formControlName="firstName" type="text">

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

</div>
```

Only display validation errors if user has interacted with the form
- When the user changes field value, the control is marked as "dirty"
- When the field loses focus, the control is marked as "touched"

Step 3: Update HTML template to display error messages



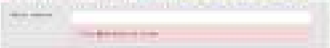
```
<label>First Name:</label>
<input formControlName="firstName" type="text">
<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required">
   First name is required
  </div>

  <div *ngIf="firstName.errors
   First Name must be at least
  </div>

</div>
```

firstName.errors object contains list of validations that failed for the form control

Step 3: Update HTML template to display error messages

Display specific error message

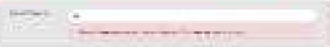
```
<label>First Name</label>
<input formControlName="firstName" type="text"/>

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required">
   First name is required
  </div>

  <div *ngIf="firstName.errors.minlength">
   First Name must be at least 2 characters long
  </div>

</div>
```



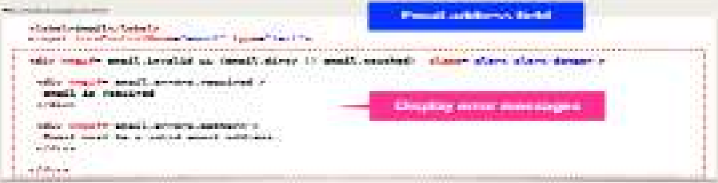
Step 3: Update HTML template to display error messages

```
<label>First Name</label>
<input formControlName="firstName" type="text">

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required">
   First name is required
  </div>

  <div *ngIf="firstName.errors.minlength">
   First Name must be at least 3 characters long
  </div>

</div>
```

Display specific error message

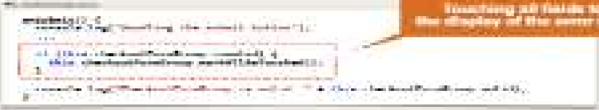


Step 3: Update HTML template to display error messages



```
<label>Email</label>
<input formControlName="email" type="text">

<div *ngIf="email.invalid && (email.dirty || email.touched)" class="alert alert-danger">

  <div *ngIf="email.errors.required">
   Email is required
  </div>

  <div *ngIf="email.errors.pattern">
   Email must be a valid email address.
  </div>

</div>
```

Step 4: Add event handler to check validation on submit



```
onSubmit() {
  console.log("Handling the submit button");
  ...
  if (this.checkoutFormGroup.invalid) {
    this.checkoutFormGroup.markAllAsTouched();
  }
  console.log("CheckoutFormGroup is valid: " + this.checkoutFormGroup.valid);
}
```

Touching all fields triggers
the display of the error messages

Checkout Form Custom Validator Rule

White Space

- Our form currently has a problem with white space
- In the required fields
  - If you enter ONLY whitespace … it passes … YIKES!
  - Should fail!

Custom Validator

- We can resolve this with a custom validator and the [illegible]
  - Check the field value
  - If it ONLY has whitespace
    - then return an error (failed)
    - else return null (passed)

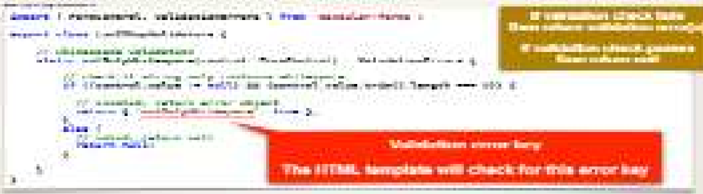
Development Process

Step-By-Step

1. Define custom validation rule
2. Specify custom validator rule for the form controls
3. Update HTML template to display error messages

Step 1: Define custom validator rule

luv2-shop-validators.ts



```
import { FormControl, ValidationErrors } from '@angular/forms';

export class Luv2ShopValidators {

    // whitespace validation
    static notOnlyWhitespace(control: FormControl) : ValidationErrors {

        // check if string only contains whitespace
        if ((control.value != null) && (control.value.trim().length === 0)) {

            // invalid, return error object
            return { 'notOnlyWhitespace': true };
        }
        else {
            // valid, return null
            return null;
        }
    }
}
```

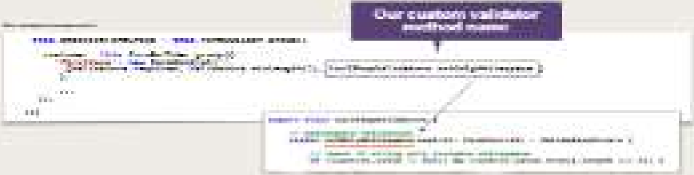
If validation check fails
then return validation error(s)

If validation check passes
then return null

Validation error key

The HTML template will check for this error key

Step 2: Specify custom validator rule for form controls

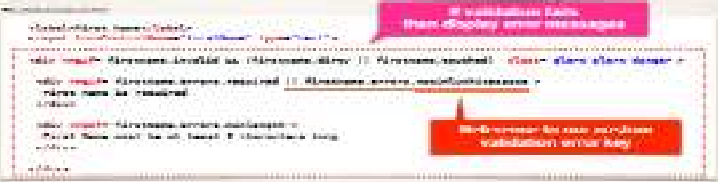


Our custom validator method name

```
this.checkoutFormGroup = this.formBuilder.group({
  customer: this.formBuilder.group({
    firstName: new FormControl('',
      [Validators.required, Validators.minLength(2), Luv2ShopValidators.notOnlyWhitespace]),
    ...
  }),
...
```

```
export class Luv2ShopValidators {

    // whitespace validation
    static notOnlyWhitespace(control: FormControl): ValidationErrors {

        // check if string only contains whitespace
        if ((control.value != null) && (control.value.trim().length === 0)) {
```

Step 3: Update HTML template to display error messages

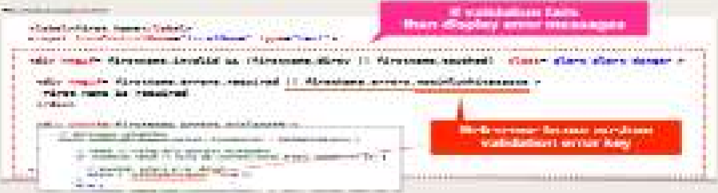


```
<label>First Name</label>
<input formControlName="firstName" type="text">

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required || firstName.errors.notOnlyWhitespace">
   First Name is required
  </div>

  <div *ngIf="firstName.errors.minlength">
   First Name must be at least 2 characters long
  </div>

</div>
```

If validation fails
then display error messages

Add check for our custom
validation error key

Step 3: Update HTML template to display error messages



```
<label>First name</label>
<input formControlName="firstName" type="text">

<div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)" class="alert alert-danger">

  <div *ngIf="firstName.errors.required || firstName.errors.notOnlyWhitespace">
    First name is required
  </div>

  <div *ngIf="firstName.errors.minlength">
```

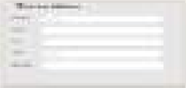
[illegible]

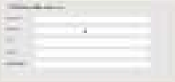
If validation fails
then display error messages

Refer to our custom
validation error key

Checkout Form Validation Shipping, Billing and Credit Card

Finish up the remaining sections





Development Process

More of the same

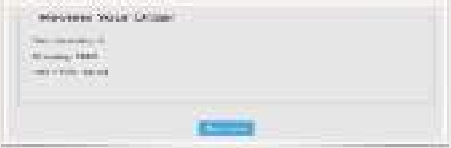
Step-By-Step

1. Specify validation rules for the form controls
2. Define Getter methods to access form controls
3. Update HTML template to display error messages

Checkout Form
Review Cart Totals

Update Checkout Form - Cart Totals

- Our form currently has a section for cart totals.
- We need to add the appropriate code to support this



Review Your Order

Total Quantity: 2

Shipping: FREE

Total Price: $0.00

Cart Service - Publishing messages/events

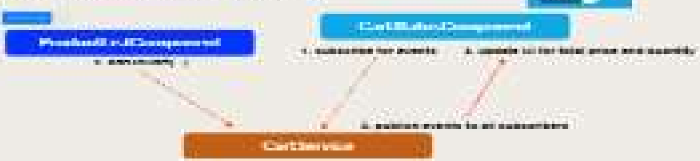
- Recall that we send messages/events to other components in our application

- For example, CartStatusComponent will subscribe to the CartService

CartService will publish messages for:

totalPrice and totalQuantity

Application Interaction



ProductListComponent

CartStatusComponent

CartService

2. addToCart()

1. subscribe for events

3. publish events to all subscribers

4. update UI for total price and quantity

CartService

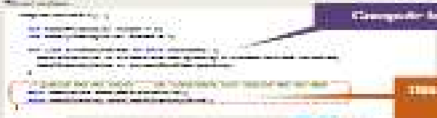
```
[illegible]
  [illegible]
  [illegible]
  [illegible]
}
```

[illegible]

[illegible]

[illegible]

CartService - Publish events



```
[illegible]
```

Compute totals

This will publish events to all subscribers

one event for ecommerce

one event for fulfilment

CartStatusComponent subscribes to CartService

Inject the CartService

Subscribe for events

When new results are received, make the assignments to update UI

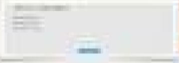
CartStatusComponent subscribes to CartService

[illegible]

When new results are received, make the assignments to update UI

Publish / Subscribe

- Similar approach for CheckoutComponent … almost
- CheckoutComponent will subscribe to events from CartService
- However, since CheckoutComponent is instantiated later in the application
  - Will miss out on previous messages
- As a result, CheckoutComponent cart totals will erroneously show as
  - 0 for total quantity
  - 0.00 for total price



Need to get a replay of the messages missed

Replay Messages

- Similar to the real world

Sorry, I am late to the meeting.
Can you tell me what I missed?

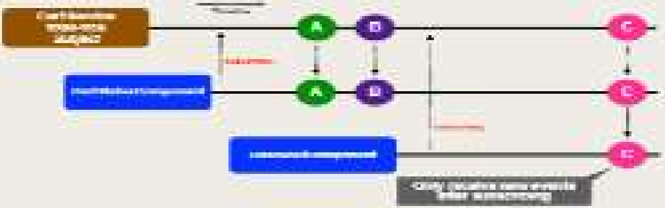
- We can get this functionality with `ReplaySubject`

ReplaySubject

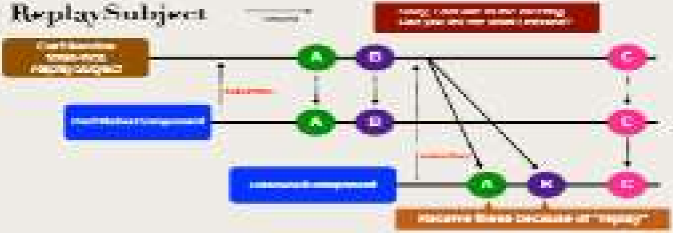
- Recall, `Subject` is used to send events to subscribers
- `ReplaySubject` is a sub-class of `Subject`
  - Will also "replay events" for new subscribers who join later
  - Keep a buffer of previous events … and send to new subscribers

Sorry, I am late to the meeting.
Can you tell me what I missed?

Subject (without replay)



ReplaySubject



ReplaySubject

```
export class CartService {

  cartItems: CartItem[] = [];

  totalPrice: Subject<number> = new ReplaySubject<number>();
  totalQuantity: Subject<number> = new ReplaySubject<number>();

}
```

Keep a buffer of previous events
Send previous events to new subscribers

ReplaySubject

- For more details on ReplaySubject, see online docs

http://www.luv2code.com/rxjs-replay-subject

But wait … there's another solution

- At first glance, ReplaySubject seems like the perfect solution
- However, for totals, we really don't need to replay the previous totals
- We are only interested in the latest total … the last computed total.
- We are only interested in the last event / message

Sorry, I am late to the meeting. What is the latest cart total?

BehaviorSubject

- BehaviorSubject is a subclass of Subject
  - Has a notion of "current value"
  - Stores the latest message / event … and sends to new subscribers

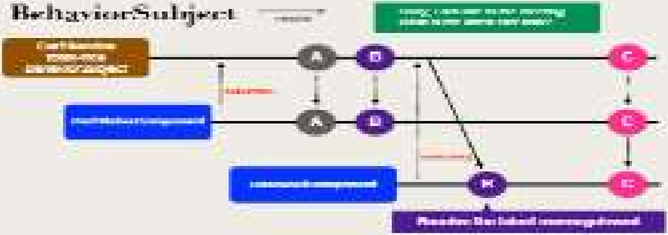
[illegible]

From the Docs

> *BehaviorSubjects are useful for representing "values over time"*
>
> *For instance, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.*

http://www.luv2code.com/rxjs-behavior-subject

BehaviorSubject



BehaviorSubject

```
export class CartService {

  cartItems: CartItem[] = [];

  totalPrice: Subject<number> = new BehaviorSubject<number>(0);
  totalQuantity: Subject<number> = new BehaviorSubject<number>(0);

}
```

Set initial value

totalPrice = 0
totalQuantity = 0

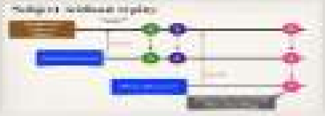
BehaviorSubject

- For more details on BehaviorSubjects, see online docs:

http://www.luv2code.com/rxjs-behavior-subject

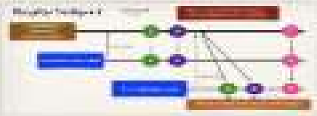
Recap

Name	Description
Subject	• Does not keep a buffer of previous events. • Subscriber only receives new events after they are subscribed



Recap

Name	Description
ReplaySubject	- Has a buffer of all previous events. - Once subscribed, subscriber receives a replay of all previous events.



Recap

Name	Description
BehaviorSubject	- Has a buffer of the last event - Once subscribed, subscriber receives the latest event sent prior to subscribing

BehaviorSubject

Recap

Name	Description
Subject	- Does not keep a buffer of previous events - Subscriber only receives new events after they are subscribed
ReplaySubject	- Has a buffer of all previous events - Once subscribed, subscriber receives a replay of all previous events
BehaviorSubject	- Has a buffer of the last event - Once subscribed, subscriber receives the latest event sent prior to subscribing

Development Process

New Update

1. Update the CheckoutComponent
   1. Inject CartService into CheckoutComponent
   2. In ngOnInit() method, call new method: reviewCartDetails()
   3. Add code for new method: reviewCartDetails()

2. Update the CartService
   1. Change subject to BehaviorSubject

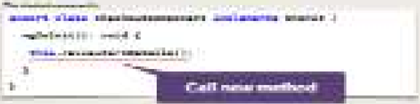
Step 1.1: Inject CartService into CheckoutForm

checkout.component.ts

```
export class CheckoutComponent implements OnInit {

  constructor(private formBuilder: FormBuilder,
              private luv2ShopFormService: Luv2ShopFormService,
              private cartService: CartService) { }

}
```

Inject CartService

Step 1.2: ngOnInit, call new method: reviewCartDetails

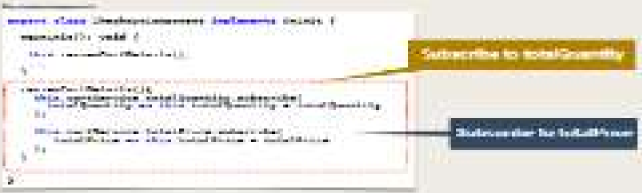


[illegible]

```
export class CheckoutComponent implements OnInit {
  ngOnInit(): void {
    this.reviewCartDetails();
  }
}
```

Call new method

Step 1.3: New method: reviewCartDetails



```
export class CheckoutComponent implements OnInit {

  ngOnInit(): void {

    this.reviewCartDetails();

  }

  reviewCartDetails() {
    this.cartService.totalQuantity.subscribe(
      totalQuantity => this.totalQuantity = totalQuantity
    );

    this.cartService.totalPrice.subscribe(
      totalPrice => this.totalPrice = totalPrice
    );
  }

}
```

Subscribe to totalQuantity

Subscribe to totalPrice

Step 2.1: Change Subject to BehaviorSubject

CartService.ts

```
export class CartService {

  cartItems: CartItem[] = [];

  totalPrice: Subject<number> = new BehaviorSubject<number>(0);
  totalQuantity: Subject<number> = new BehaviorSubject<number>(0);

}
```

Change to BehaviorSubject

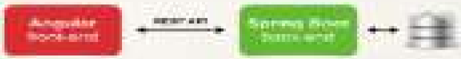
Set initial value

totalPrice = 0
totalQuantity = 0

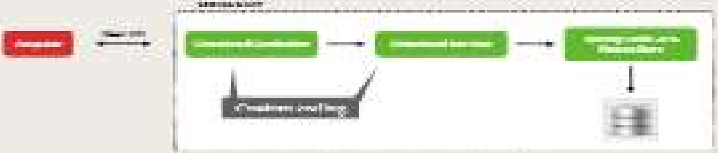
Checkout Form
Save The Order - Backend

Save The Order

- We have our shopping cart full with products … let's checkout!
- Send the order to the backend and store it in the database



Application Architecture



Custom Controller and Service

- For the architecture, we will create a custom controller and service
  - CheckoutController
  - CheckoutService

- You may wonder … why not Spring Data REST???

Why Not Spring Data REST???

- Spring Data REST is great for basic CRUD
  - We are currently using it for product catalog

- Not the best for processing the order using custom business logic
  - Generate custom tracking number
  - Save order in database
  - Other custom business logic ...

Spring Data REST
is very limited in terms of customization

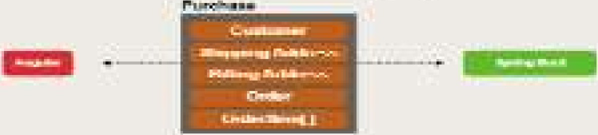
For custom business logic and processing,
create a custom controller and service

Database Diagram

Class Diagram - Entity Classes

Data Transfer Object

- Data transfer between Angular front-end and Spring Boot back-end



Purchase

Customer

Shipping Address

Billing Address

Order

OrderItem[ ]

Angular

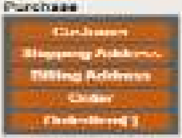
Spring Boot

REST API

- Support the POST method for checkout purchase
- Request body contains JSON for Purchase data transfer object

HTTP Method		Action
POST	/api/checkout/purchase	New purchase order

Sample JSON



Purchase
Customer
Shipping Address
Billing Address
Order
OrderItem[ ]

```
[illegible]
```

Development Process - Spring Boot

1. Run database script
2. Create entities
3. Create data transfer objects
4. Create repository
5. Create service
6. Create controller

Let's Write Some Code!

UUID

- We want a tracking number that is:
  - Hard to guess: random / unique

Google: wikipedia uuid

- UUID: Universally Unique IDentifier
  - Standardized methods for generating unique IDs
  - Available in four versions, we will use Version 4 (random)

What about Uniqueness / Collisions??

- The probability of collision is VERY low ... negligible

[illegible]

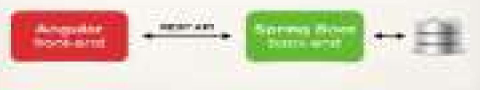
- See Collisions section of Wikipedia entry
  - Probability of a duplicate in 103 trillion version-4 UUIDs is one in a billion

- However, if you need absolute 0% of collision, you could do this instead
  - generate random UUID
  - query your db and see if UUID has been used by your app
  - if so, then repeat previous two steps until no duplicates found

Checkout Form
Save The Order - Front End

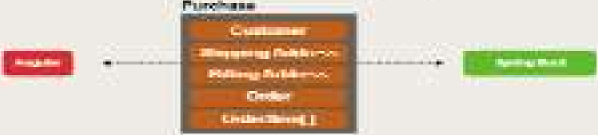
Save The Order

- The Spring Boot back end code is up and running
- Let's focus on developing Angular front-end code



Data Transfer Object

- Data transfer between Angular front-end and Spring Boot back-end



Development Process

1. Create common classes
   1. Customer, Order, OrderItem, Address, Purchase

2. Create CheckoutService
   1. Make REST API call to Spring Boot backend

3. Update CheckoutComponent
   1. Inject CheckoutService and Router
   2. Update onSubmit() method to collect form data, call CheckoutService

Let's Write Some Code!