|  |  |
| --- | --- |
| 1. RxJS | -API/Library for asynchronous programming with observable streams  -Facilitates and simplifies the processing data from multiple sources |
| 1. Reactive Development | -Code is reactive when input changes leads to automatic changes in output  -Declarative programming paradigm  -Concerned with data streams and propagation of change |
| 1. Observable   apples$.subscribe(); | -collection of events/values emitted over time  -emits notification (conveyor belt)  -created from an event/data source  -subscribe to start  HOT: starts emitting even with no subscriber  COLD: waits for subscriber |
| 1. of/from | -creator function, automatically completes and unsubscribes    -from can create an observable from any data      (HTML element, event to watch)  -document means entire HTML document    -emits ascending integers |
| 1. Operators | -functions used to transform/manipulate emitted items  -subscribe to its input observable, creates an output observable  -if more than one line, use { } and return statement |
| 1. map tap take | -transformation operator item is transformed by the function  -utility operator, useful for debugging and side effects (changing any state that does not affect its return value)  -filtering operator, completes when it equals the specified number |
| 1. “products$ | async as products”   \*ngFor=”let category of categories$ | async”  [ngClass]=”{ active: product?.id === (selectedProduct$ | async)?.id }” | -subscribes to Observable when component is initialized  -returns each emitted value  -when a new item is emitted, component is marked to be checked for changes  -unsubscribes when component is destroyed |
| 1. Error catchError(this.handleError) | -must be after any operator that could generate an error  -catchError(err => { return throwError(() => new Error(‘Could not get’))} OR throw new Error(‘Could not get’)  -EMPTY: an observable that emits no items and immediately complete  -sometimes don’t want to rethrow an error because it will propagate to our template |
| 1. Change Detection   changeDetection: ChangeDetectionStrategy.OnPush | Default  -checkAlways strategy  -every component is checked when any change is detected  OnPush  -checked when @Input changes, event emits, a bound observable emits (async pipe)  -improves performance by minimizing change detection cycle |
| 1. Procedural vs Declarative | -Assign it directly instead of calling methods  products$ = this.http.get<Product[]>(this.productsUrl) |
| 1. Mapping/Transform Array Elements | Products.map(product => ( {  …product,  searchKey: [product.name]  } as Product ))  -spead operator as shortcut  -want an object literal so add ( ) around { } |
| 1. combineLatest( [a$, b$, c$] ) | -creator/combination function  -all input must emit once  -using the latest values from each input ob  -combines each input ob into an array  -DOES NOT complete |
| 1. forkJoin( [a$, b$, c$] ) | -just like combineLatest but emits ONE time so don’t use with action stream |
| 1. pipe( withLatestFrom( ) ) | -source ob must emit first  -all observables must emit once  -emitted values combines the latest from each ob into an array  -If source completes before the other input variables, the result observables will complete without emitting anything |
| 1. Data and Action Stream | Data  -streams that will complete and won’t re-execute or react to any changes (HTTP ob)  Action  -alive and keeps emitting new actions until ob is stopped  -use combineLatest to react to actions, pipeline is also re-executed |
| 1. Subscriber | -An Observer that can unsubscribe  -From POV of subscriber, ob is read only, subscribe to react to its notifications, can’t emit anything into it  -Only the creator of Ob can emit items (of/from) |
| 17. Subject  actionSubject = new Subject<T>();  aSub = new BehaviorSubject<T>(val); | -Special type of observable that implements the Observer interface so both  -multicast (multiple subscriber)  -startsWith(val) operator  -Buffer its last emitted value to any late subscribers  -Requires a default value |
| 18. Creating an Action Stream  private productSelectedSubject = new Subject<number>();  productSelectedAction$ = this.productSelectedSubject.asObservable(); | -Good practice to encapsulate the subject variable so it can’t be accessed from outside  -Do want other parts to subscribe so define a read only (asObservable) |
| 19. Updating Data | -Before reactive, we used an array and received notifications that the data arrived from the back-end server, then assigned emitted products to that array. To add/update/delete we can simply edit the array  -For Observable, we have a data stream and can’t emit new values into a data stream  - |
| 20. scan( (acc, val) =>  (value instanceof Array) ? […val] : […acc, val], [] as Product[])  merge(a$, b$, c$) | -transformation operator when managing states  -used for encapsulating and managing state  -totaling amounts  -Creation function  -used for combining sequences of similar types to blend their emitted values |
| 21. Caching  Classic:  Private products: Product[] = []  getProducts(): Observable<Product[]> {  if (this.products) { return of(this.products); }  return this.htttp.get<Product[]>(url) | -retain retrieved data locally or reuse previously retrieved data  -run time caching is when cache data is gone once user exits the app  -improves responsiveness, reduces bandwidth, network consumption, backend server load |
| 22. shareReplay()  products$ = this.http.get<Products[]>(url)  .pipe(  shareReplay(1) | -multicast operator  -replays the defined number of emissions on subscription  -used for sharing observables, caching data, replaying emissions to late subscribers |
| 22. Part 2  Consider  -Invalidating the cache on time interval  -Allowing the user to control when data is refreshed  -Always get fresh data on update operations | -returns a Subject that shares a single subscription to the underlying source  -Takes in an optional buffer size, which is the number of items cached and replayed  -On a subscribe, it replays the specified number of emissions  -The items stay cached forever |
| 23. Higher Order Mapping Operators  xxxMap() | -automatically un/subscribes to inner observables  -flattens the result  -flattening returns the value emitted from the inner observable |
| 24. concatMap(i => of(i))  mergeMap(i => of(i))  switchMap(i => of(i)) | -think of it as a relay, concatenates their results in sequence  -think of it as a race, results is in no order  -executes inner observaables in parallel  -unsubs the prior and switches to a new inner observable  -only subs to one at a time  -if inner completes before next, then merge |
| 25. Combining streams  Get it All vs Just in Time | Get it All  -gets all the data from related data sources as streams and then finds the related items within those data streams as needed |
|  |  |
|  |  |
|  |  |