Brownfield Companion

Integrating URL-First Architecture

Document Type: Migration Guide

Audience: Developers with existing Angular 13 applications

Companion To: The VVRoom Angular Textbook (Greenfield)

Angular Version: 13.x (NgModule-based architecture)

Table of Contents

Table of Contents

Brownfield Companion: Integrating URL-First Architecture

> **Document Type:** Migration Guide **Audience:** Developers with existing Angular 13 applications **Companion To:** The Vroom Angular Textbook (Greenfield) **Angular Version:** 13.x (NgModule-based architecture)

Angular Version Compatibility

> This guide is written for **Angular 13** applications using the **NgModule-based architecture** (the pattern used before standalone components became the default in Angular 15+).

Angular Version	Compatibility	Notes
13.x	✅ Full	This guide's target version
14.x	✅ Full	NgModules still default
15.x	⚠️ Adapt	Standalone components optional; NgModules work
16.x+	⚠️ Adapt	See "Modern Angular Adaptations" section below

Key Angular 13 Patterns Used

This guide assumes your application uses:

- **NgModules** (`@NgModule`) for organizing code, not standalone components
- **Constructor injection** for services, not the `inject()` function
- **Class-based services** with `@Injectable({ providedIn: 'root' })`
- **RxJS 7.x** operators imported from `rxjs/operators`
- **Router with** `RouterModule.forRoot()` configuration

Modern Angular Adaptations

If you're on Angular 15+, the patterns still work but you may prefer:

```
// Angular 13 style (used in this guide)

@Component({ selector: 'app-discover', templateUrl: './discover.component.html', providers: [ResourceManagementService] }) export class DiscoverComponent { constructor( private resources: ResourceManagementService<Filters, Data>, private urlState: UrlStateService ) {} }



// Angular 15+ style (standalone components) @Component({ selector: 'app-discover', standalone: true, imports: [CommonModule, BasicResultsTableComponent], templateUrl: './discover.component.html', providers: [ResourceManagementService] }) export class DiscoverComponent { private resources = inject(ResourceManagementService<Filters, Data>); private urlState = inject(UrlStateService); }
```

> The URL-First concepts and service implementations are **identical** regardless of Angular version—only the component declaration syntax differs.

RxJS Import Patterns (Angular 13)

Angular 13 uses RxJS 7.x. Import operators from `rxjs/operators`:

```
// Correct for Angular 13 / RxJS 7.x

import { Observable, BehaviorSubject, Subject } from 'rxjs'; import { map, filter, distinctUntilChanged, takeUntil, switchMap } from 'rxjs/operators';



// Usage with pipe() this.urlState.watchParams().pipe( map(params => this.mapper.fromUrlParams(params)), distinctUntilChanged((a, b) => JSON.stringify(a) === JSON.stringify(b)), takeUntil(this.destroy$) ).subscribe(filters => { this.filters = filters; });
```

Overview

> The Vroom textbook teaches URL-First architecture from scratch. This companion addresses a different challenge: **How do I retrofit these patterns into an existing application?**

Brownfield development is messier than greenfield. You have:

- Existing state management (possibly NgRx, services, or component state)
- Tables and pickers that already work
- Users who expect current behavior to continue
- Technical debt that can't all be addressed at once

> This guide provides a **phased migration path** that lets you adopt URL-First incrementally without breaking your application.

Table of Contents

Part 1: URL-First State Management Integration

Assessment: Where Does Your State Live?

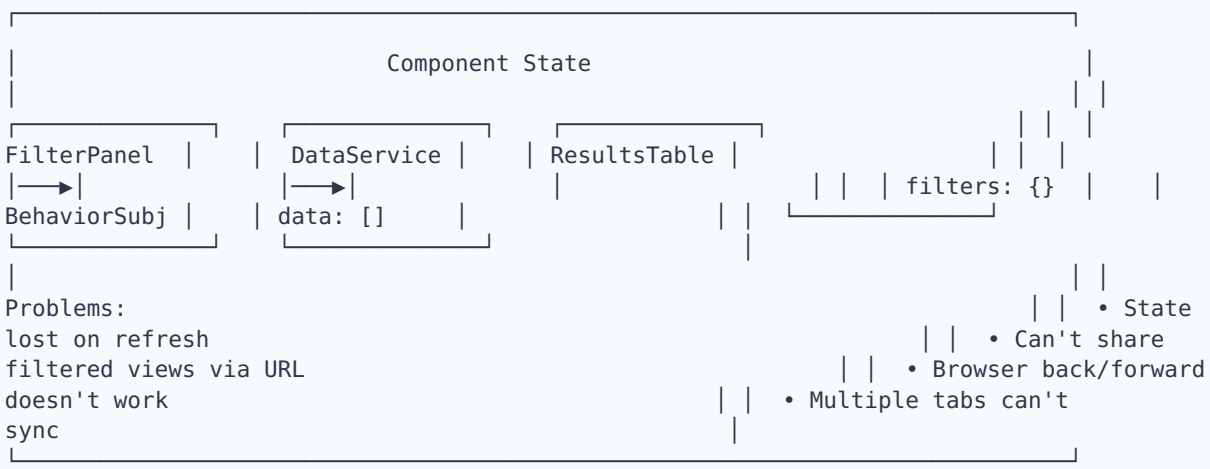
Before migrating, understand your current state architecture.

State Location Audit

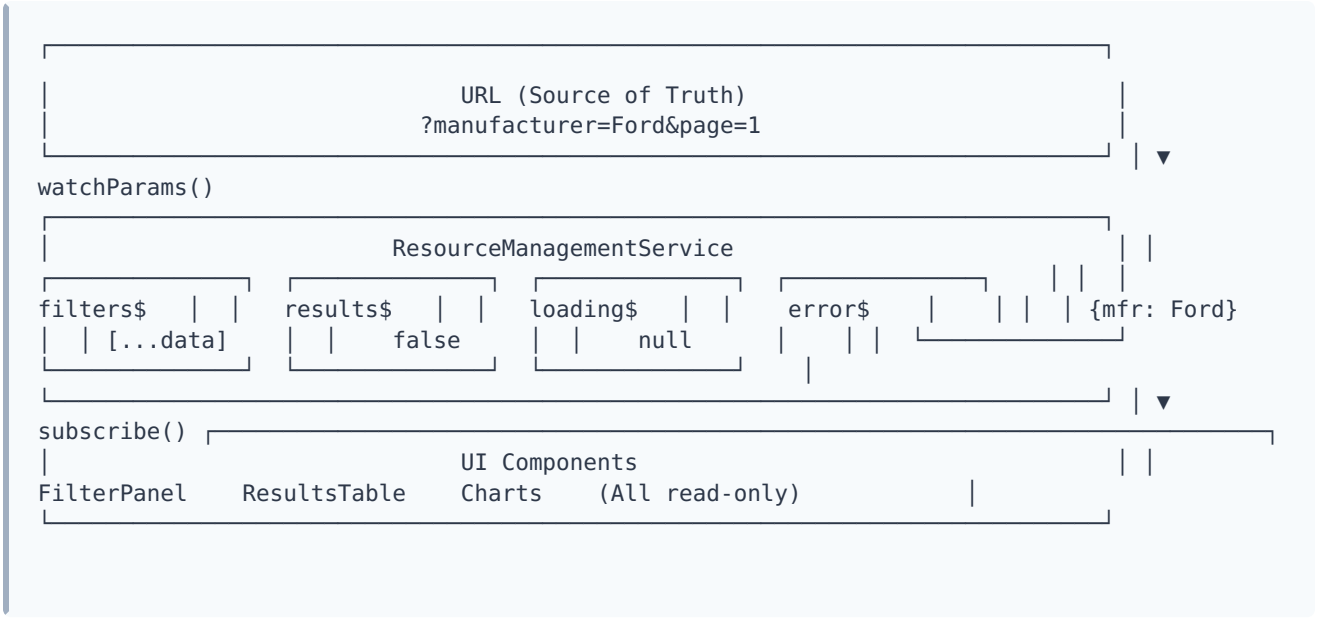
Question	If Yes...
Do components have local properties for filters/data?	Component-level state (easiest to migrate)
Do you have services with `BehaviorSubject` for filters?	Service-level state (moderate migration)
Do you use NgRx or similar for filter/results state?	Store-based state (consider keeping for complex scenarios)
Can users bookmark/share filtered views?	You may already have partial URL state
Do you have pop-out windows or multi-tab sync?	URL-First provides this for free

State Flow Diagram: Before vs After

Before (Typical Brownfield):



After (URL-First):



The Migration Strategy

The Strangler Fig Pattern

Don't rewrite everything at once. Instead, wrap old functionality with new patterns and gradually migrate:

```
Phase 1: Add URL-First services alongside existing state

Phase 2: Migrate one feature to URL-First (pilot) Phase 3: Migrate remaining features
incrementally Phase 4: Remove legacy state management
```

Risk Mitigation

Risk	Mitigation
Breaking existing functionality	Feature flags to toggle URL-First on/off
Regression in edge cases	Parallel testing: old and new side-by-side
Team unfamiliarity	Pilot with one feature before broad rollout
Performance regression	Profile before/after; URL parsing is cheap

Step 1: Install the URL State Service

The foundation of URL-First is `UrlStateService`. Add it to your application.

Create the Service

Create `src/app/framework/services/url-state.service.ts`:

```
// src/app/framework/services/url-state.service.ts
// URL-First state management foundation

import { Injectable, NgZone } from '@angular/core'; import { Router, Params,
NavigationEnd } from '@angular/router'; import { Observable, BehaviorSubject } from 'rxjs';
import { map, distinctUntilChanged, filter } from 'rxjs/operators';

/**
  • Domain-agnostic URL state management service
 *
  • Provides bidirectional synchronization between application state
  • and URL query parameters. The URL serves as the single source of truth.
 *
  • KEY DESIGN DECISIONS:
 *
  • 1. Uses Router.events instead of ActivatedRoute.queryParams because this
  •    is a root-level singleton. ActivatedRoute at root level doesn't receive
  •    query param updates from child routes (like /discover).
 *
  • 2. Uses BehaviorSubject to provide:
```

```
  •    - Synchronous access to current params via getParams()



  •    - Observable stream for reactive updates via watchParams()



  •    - Immediate value emission to new subscribers

 */
@Injectable({ providedIn: 'root' }) export class UrlStateService { private paramsSubject =
new BehaviorSubject<Params>({}); public params$: Observable<Params> =
this.paramsSubject.asObservable();



constructor( private router: Router, private ngZone: NgZone ) { this.initializeFromRoute();
this.watchRouteChanges(); }



/**



  • Get current URL query parameters synchronously

   */
getParams<TParams = Params>(): TParams { return this.paramsSubject.value as TParams; }



/**



  • Update URL query parameters (shallow merge)



  • Use null to remove a parameter

   */
async setParams<TParams = Params>( params: Partial<TParams>, replaceUrl = false ):
Promise<boolean> { const currentParams = this.paramsSubject.value; const mergedParams =
{ ...currentParams };
```

```
Object.keys(params).forEach(key => { const value = (params as any)[key]; if (value === null
|| value === undefined) { delete mergedParams[key]; } else { mergedParams[key] =
value; } });



return await this.router.navigate([], { queryParams: mergedParams, replaceUrl,
queryParamsHandling: '' }); }



/**



  • Watch URL query parameters as an observable stream


   */
watchParams<TParams = Params>(): Observable<TParams> { return this.params$.pipe( map(params
=> params as TParams), distinctUntilChanged((a, b) => JSON.stringify(a) ===
JSON.stringify(b)) ); }



/**



  • Clear all URL query parameters


   */
async clearParams(replaceUrl = false): Promise<boolean> { return this.router.navigate([],
{ queryParams: {}, replaceUrl }); }



private initializeFromRoute(): void { const params = this.extractQueryParams();
this.ngZone.run(() => { this.paramsSubject.next(params); }); }



private watchRouteChanges(): void { this.router.events .pipe( filter((event): event is
NavigationEnd => event instanceof NavigationEnd), map(() => this.extractQueryParams()),
distinctUntilChanged((a, b) => JSON.stringify(a) === JSON.stringify(b)) ) .subscribe(params
=> { this.ngZone.run(() => { this.paramsSubject.next(params); }); }); }
```

```
private extractQueryParams(): Params { const urlTree =
this.router.parseUrl(this.router.url); return urlTree.queryParams; } }
```

Verify Installation

Add a temporary test to any component:

```
import { UrlStateService } from './framework/services/url-state.service';

// In constructor constructor(private urlState: UrlStateService) {}



// In ngOnInit ngOnInit(): void { this.urlState.watchParams().subscribe(params =>
{ console.log('[URL-First] Params changed:', params); }); }
```

Navigate to `http://localhost:4200?test=123` and verify console output.

NgModule Registration (Angular 13)

The `UrlStateService` uses `providedIn: 'root'`, so it's automatically available application-wide. No module registration needed.

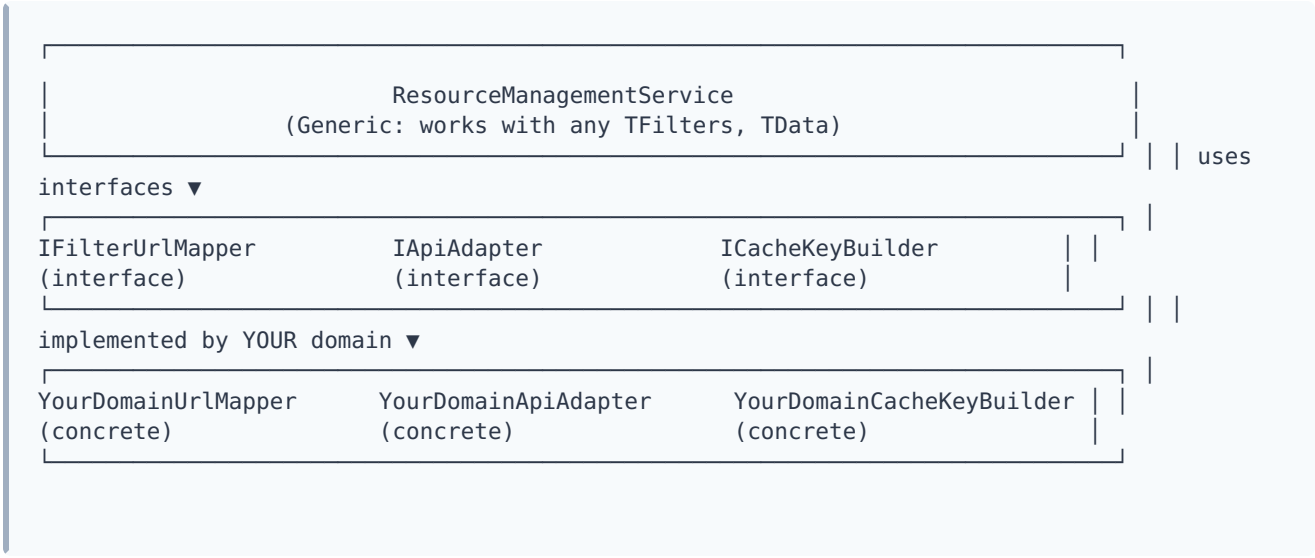
If you prefer explicit module registration:

```
// src/app/app.module.ts

import { UrlStateService } from './framework/services/url-state.service';



@NgModule({ providers: [ UrlStateService  // Optional: providedIn: 'root' handles this ] })
export class AppModule {}
```

Step 2: Create Your Domain Adapters

Adapters translate between your domain models and URL parameters.

The Adapter Pattern



Create the URL Mapper Interface

```
// src/app/framework/models/resource-management.interface.ts

import { Params } from '@angular/router'; import { Observable } from 'rxjs';



/**



  • Adapter for mapping filters to/from URL parameters

 */
export interface IFilterUrlMapper<TFilters> { /**



  • Convert domain filters to URL query parameters

   */
toUrlParams(filters: TFilters): Params;



/**



  • Convert URL query parameters to domain filters

   */
fromUrlParams(params: Params): TFilters;



/**



  • Extract highlight filters from URL parameters (optional)

   */
extractHighlights?(params: Params): any; }
```

```
/**



  • Adapter for fetching data from API


 */
export interface IApiAdapter<TFilters, TData, TStatistics = any> { fetchData( filters:
TFilters, highlights?: any ): Observable<ApiAdapterResponse<TData, TStatistics>>; }



export interface ApiAdapterResponse<TData, TStatistics = any> { results: TData[]; total:
number; statistics?: TStatistics; }
```

Example: Migrating an Existing Filter Model

> **Before (your existing code):**

```
// Your existing filter service

export interface ProductFilters { category: string; minPrice: number; maxPrice: number;
inStock: boolean; page: number; pageSize: number; }
```

> **After (create an adapter):**

```
// src/app/domains/products/adapters/product-url-mapper.ts


import { Params } from '@angular/router'; import { IFilterUrlMapper } from '../../../
framework/models/resource-management.interface'; import { ProductFilters } from '../models/
product-filters.model';



export class ProductUrlMapper implements IFilterUrlMapper<ProductFilters> {



toUrlParams(filters: ProductFilters): Params { const params: Params = {};



// Only include non-default values if (filters.category) params['category'] =
filters.category; if (filters.minPrice > 0) params['minPrice'] = String(filters.minPrice);
if (filters.maxPrice < 10000) params['maxPrice'] = String(filters.maxPrice); if
(filters.inStock) params['inStock'] = 'true'; if (filters.page > 1) params['page'] =
String(filters.page); if (filters.pageSize !== 20) params['size'] =
String(filters.pageSize);



return params; }



fromUrlParams(params: Params): ProductFilters { return { category: params['category'] ||
'', minPrice: params['minPrice'] ? Number(params['minPrice']) : 0, maxPrice:
params['maxPrice'] ? Number(params['maxPrice']) : 10000, inStock: params['inStock'] ===
'true', page: params['page'] ? Number(params['page']) : 1, pageSize: params['size'] ?
Number(params['size']) : 20 }; } }
```

Example: Wrapping Your Existing API Service

> **Before (your existing API service):**

```
@Injectable({ providedIn: 'root' })

export class ProductApiService { constructor(private http: HttpClient) {}



searchProducts(filters: ProductFilters): Observable<ProductResponse> { return
this.http.get<ProductResponse>('/api/products', { params: this.buildParams(filters) }); } }
```

After (create an adapter that wraps it):

```
// src/app/domains/products/adapters/product-api.adapter.ts


import { Injectable } from '@angular/core'; import { Observable } from 'rxjs'; import
{ map } from 'rxjs/operators'; import { IApiAdapter, ApiAdapterResponse } from '../../../
framework/models/resource-management.interface'; import { ProductFilters } from '../models/
product-filters.model'; import { Product } from '../models/product.model'; import
{ ProductApiService } from '../services/product-api.service'; // YOUR EXISTING SERVICE



@Injectable({ providedIn: 'root' }) export class ProductApiAdapter implements
IApiAdapter<ProductFilters, Product> {



constructor(private productApi: ProductApiService) {}



fetchData(filters: ProductFilters): Observable<ApiAdapterResponse<Product>> { // Delegate
to your existing API service return
this.productApi.searchProducts(filters).pipe( map(response => ({ results:
response.products, total: response.totalCount, statistics: response.facets //
optional })) ); } }
```

Key insight: You don't replace your existing API service. You **wrap** it with an adapter that conforms to the framework interface.

Step 3: Wire Up ResourceManagementService

Install the Service

Create `src/app/framework/services/resource-management.service.ts` (see Section 306 in main textbook for full implementation).

The key points for brownfield integration:

```
@Injectable() // NOT providedIn: 'root' — component-level injection

export class ResourceManagementService<TFilters, TData, TStatistics = any> implements
OnDestroy {



// Observable streams for components to subscribe to public readonly filters$:
Observable<TFilters>; public readonly results$: Observable<TData[]>; public readonly
loading$: Observable<boolean>; public readonly error$: Observable<Error | null>;



constructor( private readonly urlState: UrlStateService, @Inject(DOMAIN_CONFIG) private
readonly domainConfig: DomainConfig<TFilters, TData, TStatistics> ) { // Initialize from
current URL this.initializeFromUrl(); // Watch for URL changes this.watchUrlChanges(); }



/**



  • Update filters → Updates URL → Triggers data fetch


   */

updateFilters(partial: Partial<TFilters>): void { // ... see full implementation
this.urlState.setParams(newUrlParams); } }
```

Create the Domain Config

```
// src/app/domains/products/product-domain.config.ts


import { InjectionToken } from '@angular/core'; import { DomainConfig } from '../../
framework/models/domain-config.interface'; import { ProductFilters } from './models/
product-filters.model'; import { Product } from './models/product.model'; import
{ ProductUrlMapper } from './adapters/product-url-mapper'; import { ProductApiAdapter }
from './adapters/product-api.adapter';



export const PRODUCT_DOMAIN_CONFIG: DomainConfig<ProductFilters, Product> = { domainKey:
'products', displayName: 'Product Catalog', urlMapper: new ProductUrlMapper(), apiAdapter:
null, // Injected at runtime (see provider) tableConfig: { columns: [ { field: 'name',
header: 'Product Name', sortable: true }, { field: 'category', header: 'Category',
sortable: true }, { field: 'price', header: 'Price', sortable: true, width: '100px' } ],
dataKey: 'id' } };



// Provider factory export function productDomainConfigFactory(apiAdapter:
ProductApiAdapter): DomainConfig<ProductFilters, Product> { return
{ ...PRODUCT_DOMAIN_CONFIG, apiAdapter }; }



export const DOMAIN_CONFIG = new InjectionToken<DomainConfig<any, any>>('DOMAIN_CONFIG');



export const PRODUCT_DOMAIN_PROVIDERS = [ { provide: DOMAIN_CONFIG, useFactory:
productDomainConfigFactory, deps: [ProductApiAdapter] }, ResourceManagementService ];
```

Step 4: Migrate Components Incrementally

The Strangler Pattern for Components

> **Phase A: Side-by-side**

```
@Component({

template: <!-- OLD: Your existing filter panel --> <app-old-filter-panel *ngIf="!useUrlFirst" [filters]="legacyFilters" (filtersChange)="onLegacyFilterChange($event)"> </app-old-filter-panel>

<!-- NEW: URL-First version --> <app-new-filter-panel *ngIf="useUrlFirst" [filters]="resources.filters$ | async" (filtersChange)="resources.updateFilters($event)"> </app-new-filter-panel>

}) export class DiscoverComponent { useUrlFirst = false; // Feature flag }
```

Phase B: Full migration

```
@Component({

providers: [...PRODUCT_DOMAIN_PROVIDERS], template: <app-filter-panel [filters]="resources.filters$ | async" (filtersChange)="resources.updateFilters($event)"> </app-filter-panel>

<app-results-table [data]="resources.results$ | async" [loading]="resources.loading$ | async"> </app-results-table>

}) export class ProductDiscoverComponent { constructor(public resources: ResourceManagementService<ProductFilters, Product>) {} }
```

Migration Checklist Per Component

- ☐ Identify all state properties (filters, data, loading, etc.)
- ☐ Determine which state should be URL-persisted
- ☐ Create domain adapter if not exists
- ☐ Add `ResourceManagementService` provider
- ☐ Replace direct state with observable subscriptions

- ☐ Replace state mutations with `updateFilters()`
- ☐ Test: page refresh preserves state
- ☐ Test: browser back/forward works
- ☐ Test: URL can be copied and shared

Common Brownfield Challenges

Challenge 1: "I have NgRx for state management"

> **Solution:** URL-First and NgRx can coexist.

```
// Use NgRx for complex state (shopping cart, user session)

// Use URL-First for filter/search state (shareable, bookmarkable)



@Component({ providers: [ResourceManagementService] }) export class SearchComponent { //
URL-First for search/filter state constructor( public resources:
ResourceManagementService<SearchFilters, SearchResult>, private store: Store<AppState> //
NgRx for other state ) {}



// Cart state from NgRx cart$ = this.store.select(selectCart);



// Search state from URL results$ = this.resources.results$; }
```

Challenge 2: "My filter values are complex objects"

> **Solution:** Serialize/deserialize in your URL mapper.

```
export class ComplexUrlMapper implements IFilterUrlMapper<ComplexFilters> {


toUrlParams(filters: ComplexFilters): Params { return { // Simple values search:
filters.search,



// Arrays: comma-separated categories: filters.categories?.join(',') || null,



// Objects: JSON (base64 if needed) dateRange: filters.dateRange ? $
{filters.dateRange.start},${filters.dateRange.end} : null,



// Nested: flatten with prefixes 'price.min': filters.priceRange?.min, 'price.max':
filters.priceRange?.max }; }



fromUrlParams(params: Params): ComplexFilters { const dateRange =
params['dateRange']?.split(','); return { search: params['search'] || '', categories:
params['categories']?.split(',').filter(Boolean) || [], dateRange: dateRange ? { start:
dateRange[0], end: dateRange[1] } : null, priceRange: { min: params['price.min'] ?
Number(params['price.min']) : 0, max: params['price.max'] ? Number(params['price.max']) :
10000 } }; } }
```

Challenge 3: "Some state shouldn't be in the URL"

> **Solution:** Only URL-map the shareable state.

```
URL State (Shareable):          Component State (Local):

├── search query                ├── which accordion is expanded ├── selected
filters            ├── tooltip visibility ├── current page               ├── form
validation state ├── sort column/direction       └── temporary draft values └── selected
item IDs
```

```
@Component({...})

export class DiscoverComponent { // URL-managed (survives refresh, shareable) filters$ =
this.resources.filters$;



// Component-managed (local, transient) isAdvancedOpen = false; tooltipVisible = false;
draftSearch = ''; }
```

Challenge 4: "I need to migrate 20+ components"

> **Solution:** Prioritize by impact.

Priority	Criteria	Action
High	Main search/filter pages	Migrate first (highest user impact)
Medium	Secondary list views	Migrate after validation
Low	Admin screens, modals	Migrate opportunistically
Skip	Static pages, forms	No URL state needed

Part 2: Framework Component Conversion

Assessment: What Components Do You Have?

Component Inventory

Component Type	Signs You Have It	Framework Replacement
Data Table	`<p-table>`, `<mat-table>`, custom `*ngFor` table	`BasicResultsTableComponent`
Multi-select Picker	Dropdown/modal with checkboxes, "Apply" button	`BasePickerComponent`
Chart Display	Plotly, Chart.js, or similar	`BaseChartComponent`
Filter Panel	Form with multiple filter inputs	`QueryPanelComponent`
Active Filter Chips	Tags showing current filters	`InlineFiltersComponent`

Why Convert?

Converting existing components to framework components provides:

- **URL Synchronization** - Selections persist in URL automatically
- **Consistent UX** - Unified behavior across your application
- **Less Code** - Configuration replaces implementation
- **Pop-out Support** - Works in pop-out windows with no extra code

---

Setting Up the Framework Module (Angular 13)

Before using framework components, create the module that declares and exports them:

```
// src/app/framework/framework.module.ts


import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';



// PrimeNG modules (or your UI library) import { TableModule } from 'primeng/table'; import
{ ButtonModule } from 'primeng/button'; import { CheckboxModule } from 'primeng/checkbox';
import { InputTextModule } from 'primeng/inputtext'; import { SkeletonModule } from
'primeng/skeleton';



// Framework components import { BasicResultsTableComponent } from './components/basic-
results-table/basic-results-table.component'; import { BasePickerComponent } from './
components/base-picker/base-picker.component';



@NgModule({ declarations: [ BasicResultsTableComponent, BasePickerComponent ], imports:
[ CommonModule, FormsModule, TableModule, ButtonModule, CheckboxModule, InputTextModule,
SkeletonModule ], exports: [ // Export components for use in feature modules
BasicResultsTableComponent, BasePickerComponent ] }) export class FrameworkModule {}
```

Feature modules import `FrameworkModule` to access these components:

```
// src/app/features/products/products.module.ts

@NgModule({ imports: [ CommonModule, FrameworkModule  // <-- Provides
BasicResultsTableComponent, etc. ], declarations: [ProductDiscoverComponent] }) export
class ProductsModule {}
```

Converting Tables to BasicResultsTable

Before (Typical Custom Table)

```
@Component({

template:  <p-table
[value]="products" [loading]="loading" [paginator]="true" [rows]="pageSize" [tota
lRecords]="totalCount" (onLazyLoad)="loadData($event)"> <ng-template
pTemplate="header"> <tr> <th pSortableColumn="name">Name <p-sortIcon
field="name"></p-sortIcon></th> <th pSortableColumn="price">Price <p-sortIcon
field="price"></p-sortIcon></th> <th>Category</th> </tr> </ng-template> <ng-
template pTemplate="body" let-product> <tr> <td>{{ product.name }}</td>
<td>{{ product.price | currency }}</td> <td>{{ product.category }}</td> </tr> </
ng-template> </p-table>

})
export class ProductTableComponent implements OnInit {
products: Product[] = [];
loading = false;
totalCount = 0;
pageSize = 20;

constructor(private productService: ProductService) {}

ngOnInit() {
this.loadData({ first: 0, rows: 20 });
}

loadData(event: any) {
this.loading = true;
this.productService.search({
page: event.first / event.rows + 1,
size: event.rows,
sort: event.sortField,
sortOrder: event.sortOrder
}).subscribe(response => {
this.products = response.products;
this.totalCount = response.total;
this.loading = false;
});
}
}
```

After (Using BasicResultsTable)

> **Step 1: Create the table configuration**

```
// src/app/domains/products/config/table.config.ts


import { TableConfig } from '../../../framework/models/table-config.interface'; import
{ Product } from '../models/product.model';



export const PRODUCT_TABLE_CONFIG: TableConfig<Product> = { columns: [ { field: 'name',
header: 'Name', sortable: true }, { field: 'price', header: 'Price', sortable: true, width:
'120px' }, { field: 'category', header: 'Category', sortable: false } ], dataKey: 'id',
expandable: true, rowsPerPageOptions: [10, 20, 50, 100] };
```

Step 2: Include in domain config

```
export const PRODUCT_DOMAIN_CONFIG: DomainConfig<ProductFilters, Product> = {

domainKey: 'products', displayName: 'Products', urlMapper: new ProductUrlMapper(),
apiAdapter: null, // injected tableConfig: PRODUCT_TABLE_CONFIG };
```

Step 3: Use the framework component

```
@Component({

selector: 'app-product-discover', templateUrl: './product-discover.component.html',
providers: [...PRODUCT_DOMAIN_PROVIDERS] }) export class ProductDiscoverComponent
{ domainConfig = PRODUCT_DOMAIN_CONFIG;



constructor(public resources: ResourceManagementService<ProductFilters, Product>) { //
ResourceManagementService wires up data loading automatically } }
```

```
<!-- product-discover.component.html -->

<app-basic-results-table [domainConfig]="domainConfig"> </app-basic-results-table>
```

Step 4: Register in your feature module (Angular 13)

```
// src/app/features/products/products.module.ts


import { NgModule } from '@angular/core'; import { CommonModule } from '@angular/common';
import { FrameworkModule } from '../../framework/framework.module'; import
{ ProductDiscoverComponent } from './product-discover.component'; import
{ ProductsRoutingModule } from './products-routing.module';



@NgModule({ declarations: [ ProductDiscoverComponent ], imports: [ CommonModule,
FrameworkModule,  // Exports BasicResultsTableComponent ProductsRoutingModule ] }) export
class ProductsModule {}
```

What You Get For Free

Feature	Your Old Code	Framework Component
Pagination	Manual `loadData()`	Automatic via URL state
Sorting	Manual sort state	Automatic via URL state
Loading skeleton	Custom implementation	Built-in
Empty state	Custom implementation	Built-in
Row expansion	Custom implementation	Config-driven
Pop-out support	Not available	Built-in

Converting Pickers to BasePicker

Before (Typical Custom Picker)

```
@Component({

template: `<p-dialog [(visible)]="visible"> <p-table [value]="categories" [loading]="loading" [(selection)]="selectedCategories" [paginator]="true" [rows]="10"> <ng-template pTemplate="header"> <tr> <th style="width: 3rem"> <p-tableHeaderCheckbox></p-tableHeaderCheckbox> </th> <th>Category Name</th> </tr> </ng-template> <ng-template pTemplate="body" let-cat> <tr> <td> <p-tableCheckbox [value]="cat"></p-tableCheckbox> </td> <td>{{ cat.name }}</td> </tr> </ng-template> </p-table> <button (click)="apply()">Apply</button> </p-dialog>`

})
export class CategoryPickerComponent {
  categories: Category[] = [];
  selectedCategories: Category[] = [];
  loading = false;
  visible = false;

  @Output() selectionChange = new EventEmitter<Category[]>();

  ngOnInit() { this.loadCategories(); }

  loadCategories() {
    this.loading = true;
    this.categoryService.getAll().subscribe(cats => {
      this.categories = cats;
      this.loading = false;
    });
  }

  apply() {
    this.selectionChange.emit(this.selectedCategories);
    this.visible = false;
  }
}
```

After (Using BasePicker)

> **Step 1: Create the picker configuration**

```
// src/app/domains/products/config/category-picker.config.ts

import { PickerConfig } from '../../../framework/models/picker-config.interface'; import
{ Category } from '../models/category.model';



export const CATEGORY_PICKER_CONFIG: PickerConfig<Category> = { id: 'category-picker',



columns: [ { field: 'name', header: 'Category Name', sortable: true } ],



api: { fetchData: (params) => { // Inject your service or use a factory return
inject(CategoryService).getAll(params); }, responseTransformer: (response) => ({ results:
response.categories, total: response.total }) },



pagination: { mode: 'server', defaultPageSize: 10, pageSizeOptions: [10, 25, 50] },



selection: { urlParam: 'categories', keyGenerator: (cat) => cat.id, serializer: (items) =>
items.map(c => c.id).join(','), deserializer: (value) => value.split(',').map(id =>
({ id })) },



row: { keyGenerator: (cat) => cat.id },



showSearch: true, searchPlaceholder: 'Search categories...' };
```

Step 2: Register in picker registry

```
// In your module or app initializer

constructor(private pickerRegistry: PickerConfigRegistry)
{ this.pickerRegistry.register('category-picker', CATEGORY_PICKER_CONFIG); }
```

Step 3: Use the framework component

```
@Component({

template:  <app-base-picker configId="category-
picker" (selectionChange)="onCategoryChange($event)"> </app-base-picker>

})
export class FilterPanelComponent {



constructor(private resources: ResourceManagementService<ProductFilters, Product>) {}



onCategoryChange(event: PickerSelectionEvent<Category>) { // The picker already serializes
to URL-friendly format this.resources.updateFilters({ categories: event.urlValue || null }
as Partial<ProductFilters>); } }
```

Selection Persistence Across Pages

The framework picker automatically handles:

```
┌──────────────────────────────────────────────────────────────────────┐
│ Page 1 Loaded                                                        │ │ ☑
Electronics     Selected items: [Electronics]                          │ │ ☐
Clothing                                                             │ │ ☐
Home                                                                 │
└──────────────────────────────────────────────────────────────────────┘ │ │ Navigate to
page 2 ▼ ┌──────────────────────────────────────────────────────────────────────┐ │ Page
2 Loaded                                                             │ │ ☑ Sports
Selected items: [Electronics, Sports]                  │ │ ☐ Books           (Electronics
preserved even though not visible)     │ │ ☐
Toys                                                                 │
└──────────────────────────────────────────────────────────────────────┘ │ │ Navigate
back to page 1 ▼
┌──────────────────────────────────────────────────────────────────────┐ │ Page 1
Loaded                                                               │ │ ☑ Electronics
Selected items: [Electronics, Sports]                  │ │ ☐ Clothing        (Both selections
preserved!)                            │ │ ☐
Home                                                                 │
└──────────────────────────────────────────────────────────────────────┘
```

Preserving Custom Behavior

Extending Framework Components

If framework components don't cover your use case, extend them:

```
@Component({

selector: 'app-product-table', template:  <!-- Wrapper with custom header --> <div class="product-table-wrapper"> <div class="custom-header"> <button (click)="exportCsv()">Export CSV</button> <button (click)="printView()">Print</button> </div>

<!-- Framework component does the heavy lifting --> <app-basic-results-table [domainConfig]="domainConfig" (rowClick)="onProductClick($event)"> </app-basic-results-table> </div>

}) export class ProductTableWrapperComponent { @Input() domainConfig!: DomainConfig<ProductFilters, Product>;



constructor(private exportService: ExportService) {}



exportCsv() { // Your custom export logic this.exportService.exportTableToCsv(); }



onProductClick(product: Product) { // Your custom click handling this.router.navigate(['/products', product.id]); } }
```

Custom Column Rendering

For custom cell rendering, extend the table config:

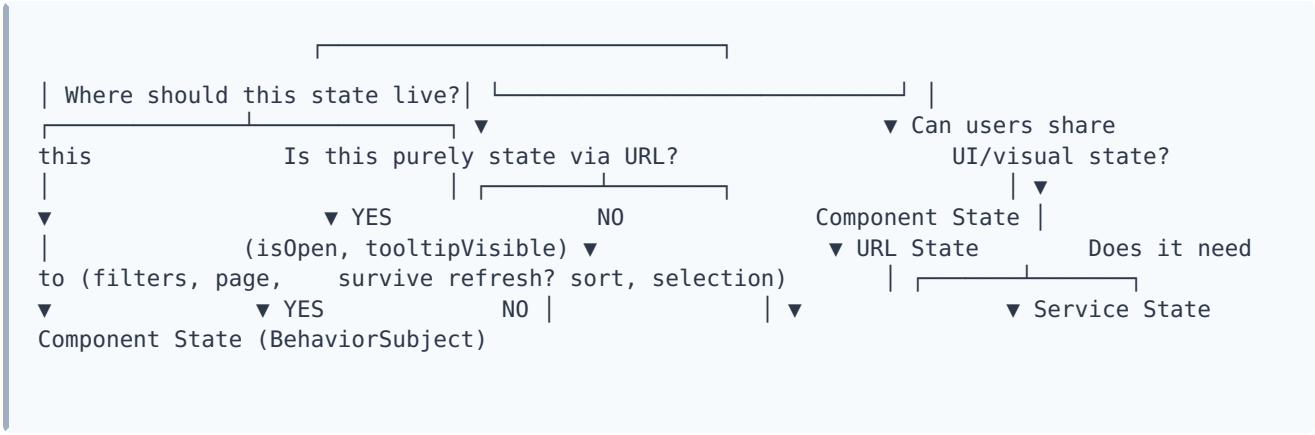
```
// Custom pipe for complex formatting

@Pipe({ name: 'productPrice' }) export class ProductPricePipe implements PipeTransform
{ transform(value: number, currency: string): string { return new Intl.NumberFormat('en-
US', { style: 'currency', currency }).format(value); } }



// In table config export const PRODUCT_TABLE_CONFIG: TableConfig<Product> = { columns:
[ { field: 'name', header: 'Name' }, { field: 'price', header: 'Price', // Custom template
reference (if supported) template: 'priceTemplate' } ] };
```

Migration Patterns Reference

Quick Reference: URL Mapper Patterns

Data Type	URL Representation	Example
String	Direct	`search=laptop`
Number	String	`page=2`
Boolean	`true` / `false` string	`inStock=true`
Array	Comma-separated	`categories=elec,home,toys`
Date	ISO string	`startDate=2024-01-15`
Date Range	Comma-separated	`dates=2024-01-01,2024-01-31`
Object	Flattened with dots	`price.min=10&price.max=100`

Quick Reference: State Decision Tree



Migration Effort Estimates

Scenario	Complexity	Typical Effort
Simple filter page → URL-First	Low	1-2 days
Complex filter page with pickers	Medium	3-5 days
NgRx integration	Medium-High	5-10 days
Full application migration	High	2-4 weeks
Table → BasicResultsTable	Low	1 day
Picker → BasePicker	Medium	2-3 days

Success Criteria

Your URL-First migration is complete when:

- ☐ Users can bookmark any filtered view
- ☐ Copying URL shares exact application state
- ☐ Browser back/forward navigates through filter history
- ☐ Page refresh preserves all filter state
- ☐ Multiple tabs can show different filter states

- ☐ Pop-out windows (if used) sync with main window
- ☐ No legacy filter state services remain

This companion guide is maintained alongside the Vroom Angular Textbook. For greenfield projects, start with the main textbook. For brownfield migrations, use this guide to incrementally adopt URL-First patterns.