FeatureDetectorService

September 4, 2026

FeatureDetectorService

Kind: Service

Source: `atloria-monorepo/apps/api/src/documentation/services/feature-detector.service.ts`

Feature Detector Service

Automatically identifies "features" from parsed codebase

Detection strategies:

1. State Machine-based: Each state machine represents a feature workflow
2. Module-based: Group by application modules (Angular modules, React feature folders)
3. Route-based: Group by route prefix (e.g., /sales/*, /inventory/*)
4. Business Context-based: Extract from marketing/process docs

`FeatureDetectorService` automatically identifies higher-level “features” from a parsed codebase so the system can organize, document, and reason about workflows and domains. It applies multiple detection strategies—state machine workflows, module/folder boundaries, route prefixes, and business-context documentation—to produce a consolidated set of feature definitions for downstream tooling (e.g., docs, navigation, analysis).

Methods

Method	Signature	Returns	Description
`detectFeatures`	`detectFeatures(options: {		

```
entities: any[]; // Parsed entities from parser-core
appMap?: any; // Application structure map
businessContext?: {
  marketing: string[];
  workflows: string[];
```

```
  features: string[];
};
```

}) | Promise<DetectedFeature[]>` | Detect features from multiple sources |

Where it refuses work

- `FeatureDetectorService` stops the work with an early return when `!appMap?.routes`.
- `FeatureDetectorService` stops the work with an early return when `nameSimilarity > 0.6`.
- `FeatureDetectorService` stops the work with an early return when `sharedComponents.length > 0`.
- `FeatureDetectorService` stops the work with an early return when `sharedStateMachines.length > 0`.
- `FeatureDetectorService` stops the work with an early return when `stateMachine.transitions.length === 0`.
- `FeatureDetectorService` stops the work with an early return when `!service.methods || service.methods.length === 0`.

Diagram

```
sequenceDiagram
  autonumber
  actor Caller
  participant FDS as FeatureDetectorService
  participant Code as Parsed Codebase/AST
  participant SM as StateMachine Strategy
  participant Mod as Module Strategy
  participant Rt as Route Strategy
  participant Biz as Business Context Strategy
  participant Out as Feature Set

  Caller->>FDS: detectFeatures(codebase, options)
  FDS->>Code: read parsed structures (AST, routes, modules)
  FDS->>SM: detectFromStateMachines(codebase)
  SM-->>FDS: features (workflow-based)
  FDS->>Mod: detectFromModules(codebase)
  Mod-->>FDS: features (module/folder-based)
  FDS->>Rt: detectFromRoutes(codebase)
  Rt-->>FDS: features (route-prefix-based)
  FDS->>Biz: detectFromBusinessDocs(contextDocs)
  Biz-->>FDS: features (domain/context-based)
  FDS->>FDS: merge/dedupe/score + resolve conflicts
```

```
  FDS-->>Out: consolidated features
  Out-->>Caller: Feature[] (with sources/metadata)
```

Usage

```
import { FeatureDetectorService } from './documentation/services/feature-detector.service';

// In NestJS you would usually inject this service; this is a simplified example.
async function runFeatureDetection() {
  const featureDetector = new FeatureDetectorService();

  // Example shape: depends on your parser output; keep it consistent with what the service exp
  const parsedCodebase = {
    stateMachines: [
      /* ... */
    ],
    modules: [
      /* Angular modules / React feature folders ... */
    ],
    routes: [
      { method: 'GET', path: '/sales/orders' },
      { method: 'POST', path: '/inventory/items' },
    ],
    docs: {
      marketing: 'Sales workflow covers quote -> order -> invoice ...',
      process: 'Inventory receiving and stock adjustments ...',
    },
  };

  const features = await featureDetector.detectFeatures(parsedCodebase, {
    strategies: ['stateMachine', 'module', 'route', 'businessContext'],
    routePrefixDepth: 1, // e.g., "/sales/*" groups under "sales"
  });

  // Use the detected features to generate docs, navigation, or analysis outputs
  for (const f of features) {
    console.log(`[feature] ${f.name}`, {
      key: f.key,
      sources: f.sources,
      confidence: f.confidence,
    });
  }
}

runFeatureDetection().catch(console.error);
```

AI Coding Instructions

- Preserve the “multi-strategy” pattern: each detector should be independent (pure-ish) and return features with source metadata so merging/deduping can be explained and debugged.
- When adding a new strategy, ensure it produces stable feature keys (deterministic naming) to avoid churn in generated docs and to make deduplication reliable.
- Be careful merging results: handle collisions (same name from different strategies) via scoring/priority rules and keep provenance (`sources`) for traceability.
- Integrate with parsers/ingestors at clear boundaries: this service should consume normalized parsed structures (AST/module/route/doc summaries), not raw filesystem or framework-specific internals.

Referenced By

- `DocumentationModule` (MODULE_PROVIDES)
- `DocumentationModule` (MODULE_EXPORTS)