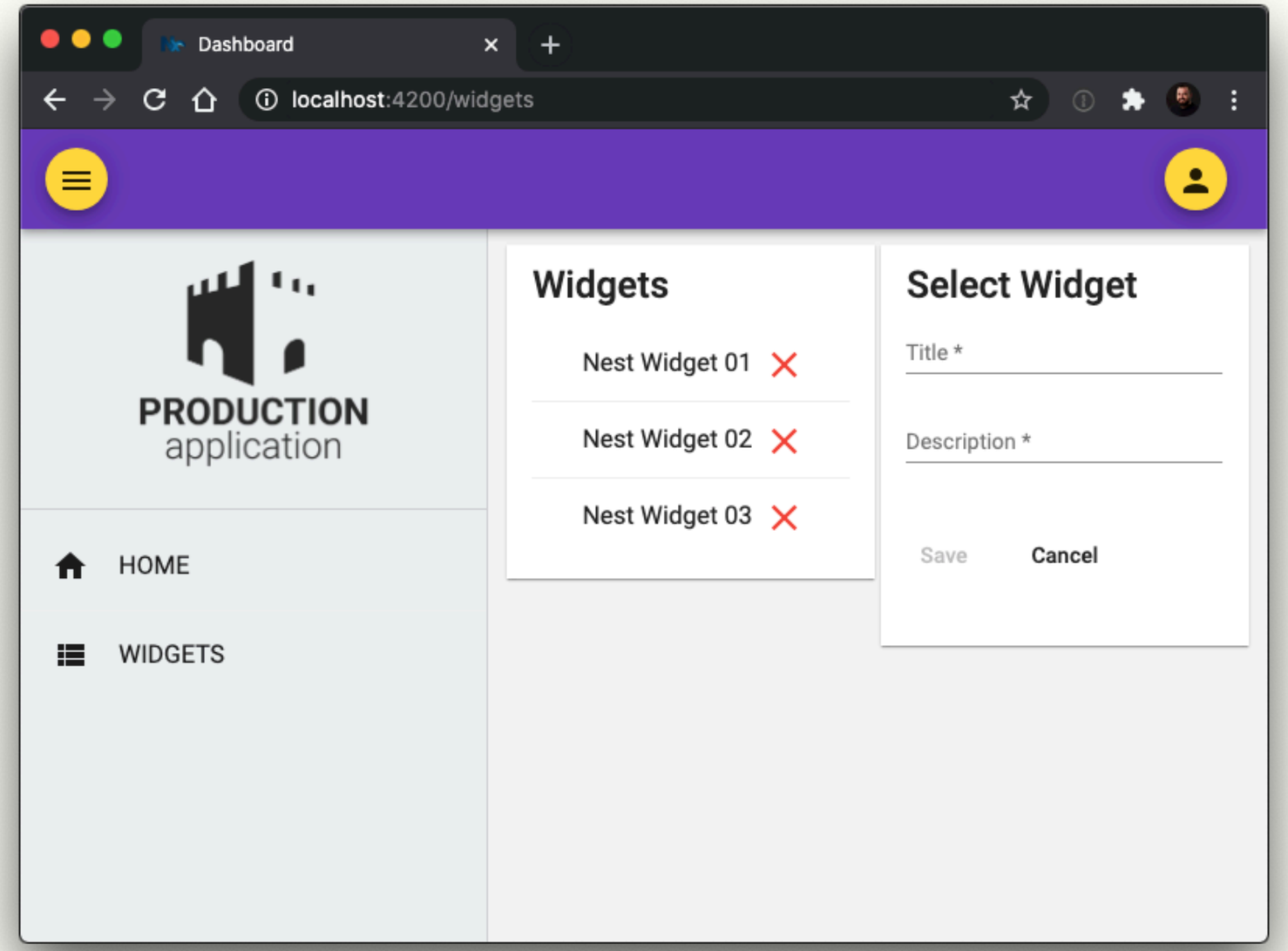
PRODUCTION ANGULAR

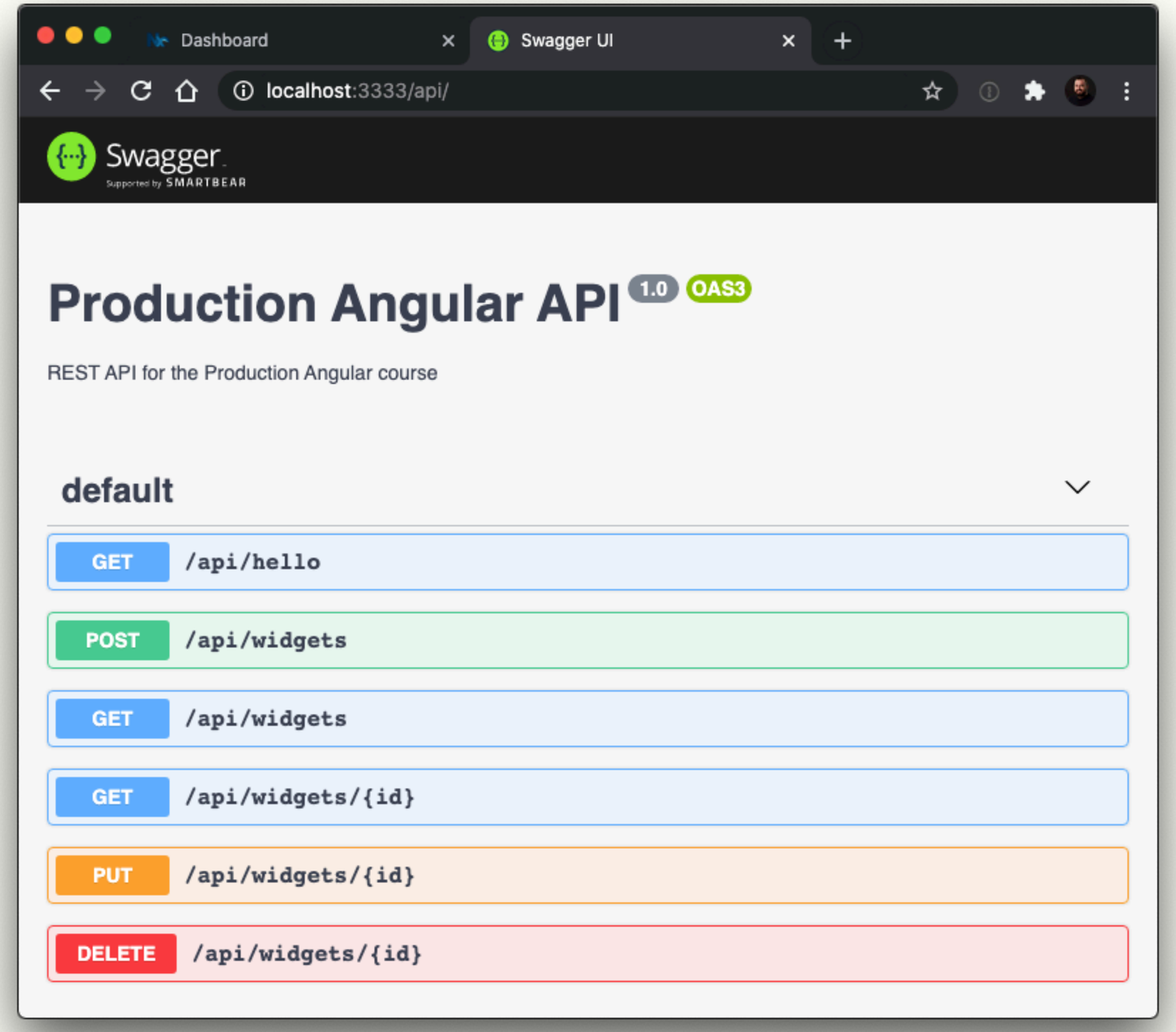
Widget

Introduction

We are going to **"hang out"**
and I am going to show you
the **patterns** and **techniques**
that I use everyday to **develop**
Angular applications for
production

https://github.com/onehungrymind/fem-production-angular


Dashboard
localhost:4200/widgets
PRODUCTION
application
HOME
WIDGETS
Widgets
Nest Widget 01
Nest Widget 02
Nest Widget 03
Nest Widget 03
Select Widget
Title *
Description *
Save
Cancel


Dashboard
Swagger UI
localhost:3333/api/
Swagger
Supported by SMARTBEAR
Production Angular API
1.0
OAS3
REST API for the Production Angular course
default
GET /api/hello
POST /api/widgets
GET /api/widgets
GET /api/widgets/{id}
PUT /api/widgets/{id}
DELETE /api/widgets/{id}

Agenda

- **Managing Complexity**
- **Make It Work**
- Angular CLI and Nx Workspaces
- Managing Multiple Applications
- Mock APIs in Minutes
- Reactive Angular
- **Make It Right**
- Writing Testable Code
- E2E Testing with Cypress
- **Make It Fast**
- Performance Analysis
- Building Angular for Production
- Deploying to Production
- **Wrap Up and Q&A**

Managing Complexity

Managing complexity
is the hardest thing
about developing
software

Complexity consists of **managing of state**, flow **control**, and **code** volume

General rules for managing complexity

Make it **work**
Make it **right**
Make it **fast**

Make it **work**
Make it **known**
Make it **right**
Make it **fast**

Code should be **fine**
grained

Code should **do one**
thing

Code should be **self**
documenting

Favor **pure, immutable**
functions

Abstractions should reduce complexity

Abstractions should reduce coupling

Abstractions should increase cohesion

Abstractions should increase portability

Refactor through promotion

Composition over inheritance

Do not confuse
convention for
repetition

Well-structured code
will naturally have a
larger surface area

Team rules for managing complexity

Be mindful over the
limitations of your
entire team and
optimize around that

Favor best practices
over introducing idioms
however clever they
may be

Consistency is better
than **righteousness**

Follow the style guide until it doesn't make sense for your situation

Tactical rules for managing complexity

Eliminate **hidden state** in functions

Eliminate **nested logic** in functions

Do not break the **Single Responsibility** Principle

Extracting to a method is one of the most effective refactoring strategies available

If you need to **clarify your code with comments** then it is *probably* too complex

It is *impossible* to write **good tests** for **bad code**

Specific rules for managing complexity

Your routing table will generally describe your features

A feature will generally get a route

A route will navigate to a container component

Everything inside that container component should be a presentation component

A component should only ever do two things...

Consume just enough data to satisfy its templates

Capture user events and delegate them upwards

Components should be as thin as possible

Container components should satisfy inputs using the async pipe

Components should be oblivious to business logic

Components should be oblivious to server communication

Components should be oblivious to application state

Facades are an effective delegation layer between components and the rest of the app

Facades are for delegation only

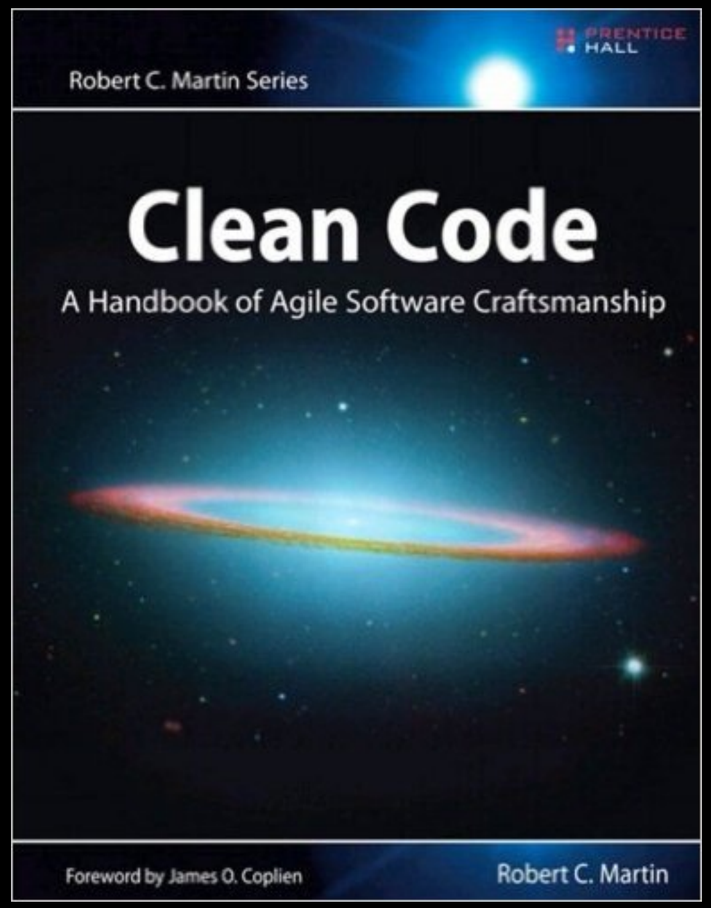
Server communication and state management should be decoupled

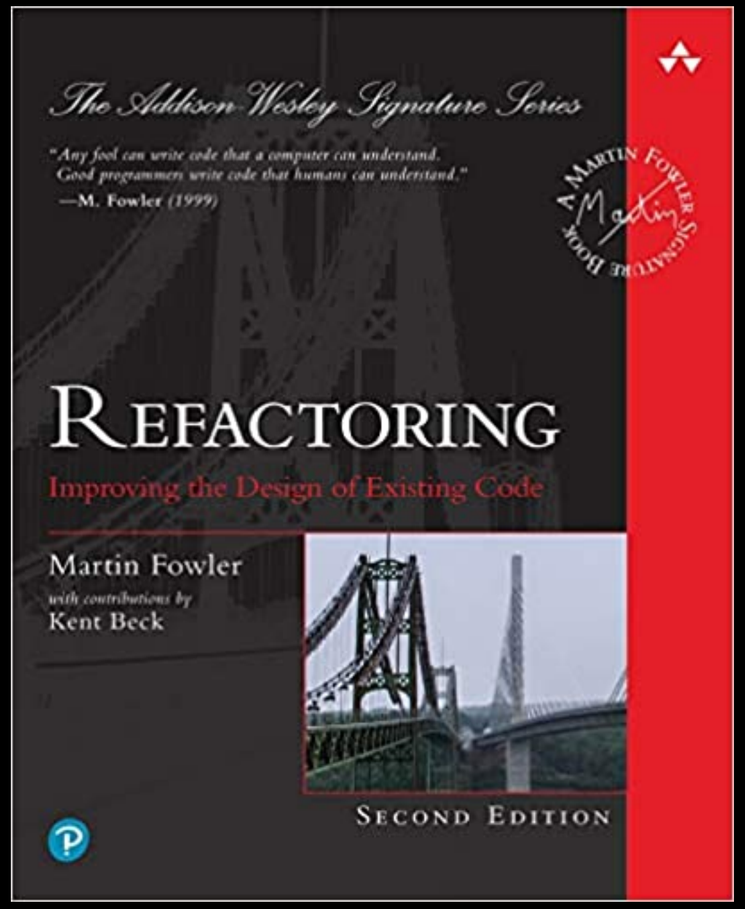
Data models should be decoupled especially inside of a monorepo with client and API projects

Do not unnecessarily optimize until you have a good reason to do so

For instance, a component should not become a lib until it is going to be used in more than one app

Resources


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


The Addison Wesley Signature Series
"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
—M. Fowler (1999)
A MARTIN FOWLER SIGNATURE BOOK
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

https://angular.io/guide/styleguide

DISCUSSION TIME!

Make It **Work**

Angular CLI and Nx Workspaces

Data **Modeling**

Nx Workspaces

Code Generation

BONUS! JSON Server

```
$ npx create-nx-workspace@10.3.2 fem-production-angular \
    --appName=dashboard \
    --preset=angular-nest \
    --npmScope=fem \
    --linter=tslint \
    --nx-cloud=false \
    --style=scss
```

Create Nx Workspace

```
nx g lib core-data --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib core-state --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib material --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g s services/widgets/widgets --project=core-data && \
nx g m routing --flat=true -m=app.module.ts && \
nx g c widgets -m app.module.ts --style=scss && \
nx g c widgets/widgets-list -m app.module.ts --style=scss && \
nx g c widgets/widget-details -m app.module.ts --style=scss && \
nx g c home -m app.module.ts --style=scss
```

Generate Code

```
nx g lib core-data --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib core-state --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib material --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g s services/widgets/widgets --project=core-data && \
nx g m routing --flat=true -m=app.module.ts && \
nx g c widgets -m app.module.ts --style=scss && \
nx g c widgets/widgets-list -m app.module.ts --style=scss && \
nx g c widgets/widget-details -m app.module.ts --style=scss && \
nx g c home -m app.module.ts --style=scss
```

Generate Code

```
nx g lib core-data --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib core-state --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib material --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g s services/widgets/widgets --project=core-data && \
nx g m routing --flat=true -m=app.module.ts && \
nx g c widgets -m app.module.ts --style=scss && \
nx g c widgets/widgets-list -m app.module.ts --style=scss && \
nx g c widgets/widget-details -m app.module.ts --style=scss && \
nx g c home -m app.module.ts --style=scss
```

Generate Code

```
nx g lib core-data --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib core-state --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib material --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g s services/widgets/widgets --project=core-data && \
nx g m routing --flat=true -m=app.module.ts && \
nx g c widgets -m app.module.ts --style=scss && \
nx g c widgets/widgets-list -m app.module.ts --style=scss && \
nx g c widgets/widget-details -m app.module.ts --style=scss && \
nx g c home -m app.module.ts --style=scss
```

Generate Code

```
nx g lib core-data --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib core-state --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g lib material --parent-module=apps/dashboard/src/app/app.module.ts --routing --style=scss && \
nx g s services/widgets/widgets --project=core-data && \
nx g m routing --flat=true -m=app.module.ts && \
nx g c widgets -m app.module.ts --style=scss && \
nx g c widgets/widgets-list -m app.module.ts --style=scss && \
nx g c widgets/widget-details -m app.module.ts --style=scss && \
nx g c home -m app.module.ts --style=scss
```

Generate Code

DEMO TIME!

Managing Multiple Applications

Multiple Applications

Common Functionality

Accelerated Development

DEMO TIME!

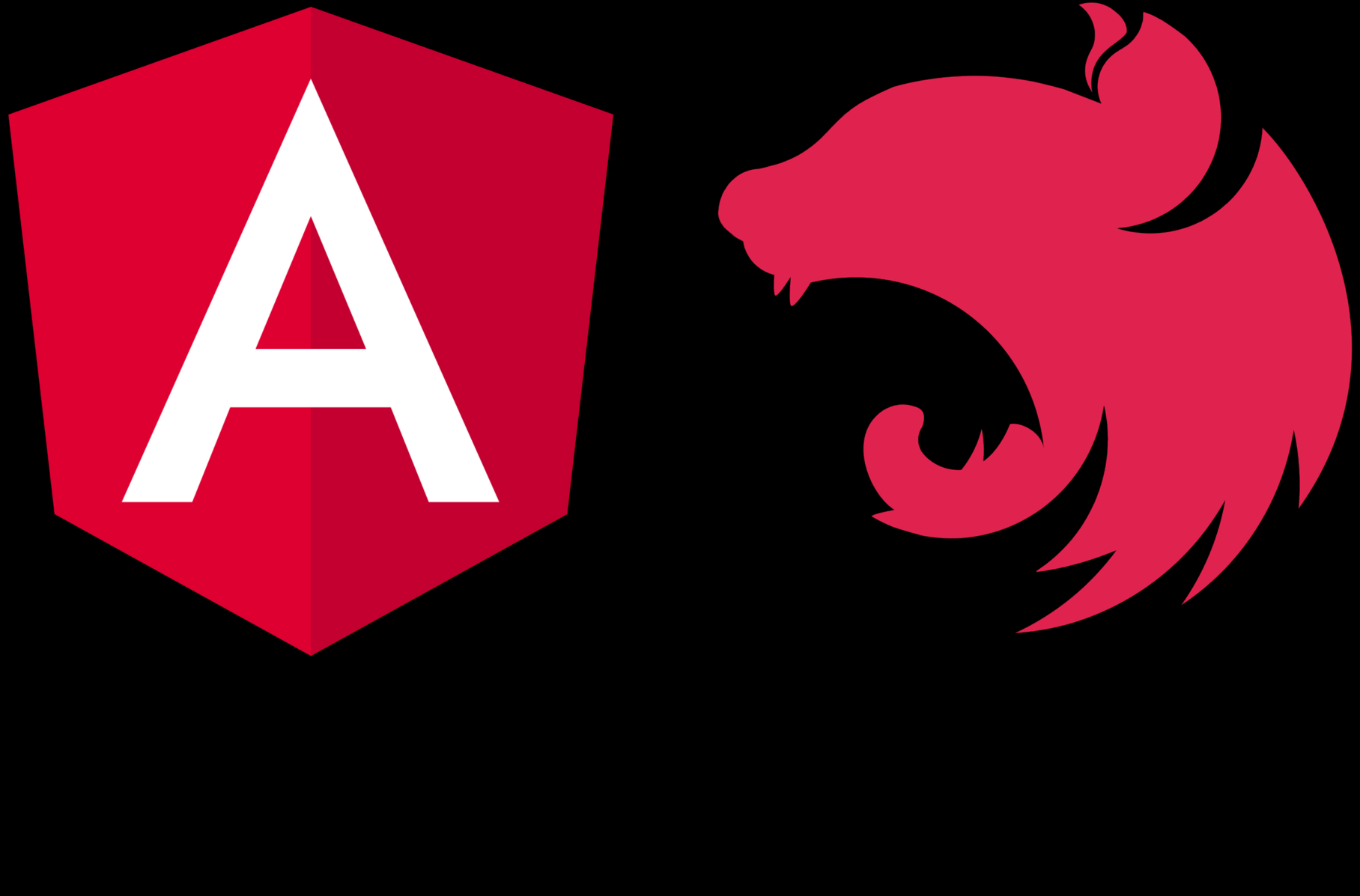
Mock APIs in Minutes

aka Hello **Nest**

Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications. The architecture is heavily inspired by Angular.

Nest provides an **out-of-the-box application architecture** which allows developers and teams to create **highly testable**, **scalable**, **loosely coupled**, and **easily maintainable** applications. The architecture is heavily inspired by Angular.

Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications. The architecture is heavily inspired by Angular.



What do we **love**
about **Angular**?

Uses TypeScript

Reactive framework

Great **tooling**

Leverages Dependency Injection

Clean abstractions

If you are **comfortable** with **Angular**, you already "know" Nest

```ts
import { Widget } from '../database/entities/widget.entity';
import { WidgetsService } from './widgets.service';

@Controller('widgets')
export class WidgetsController {
  constructor(private readonly widgetsService: WidgetsService) {}

  @Get()
  all(): Promise<Widget[]> {
    return this.widgetsService.getAll();
  }

  @Get(':id')
  find(@Param() id: string): Promise<Widget> {
    return this.widgetsService.get(id);
  }
}
```

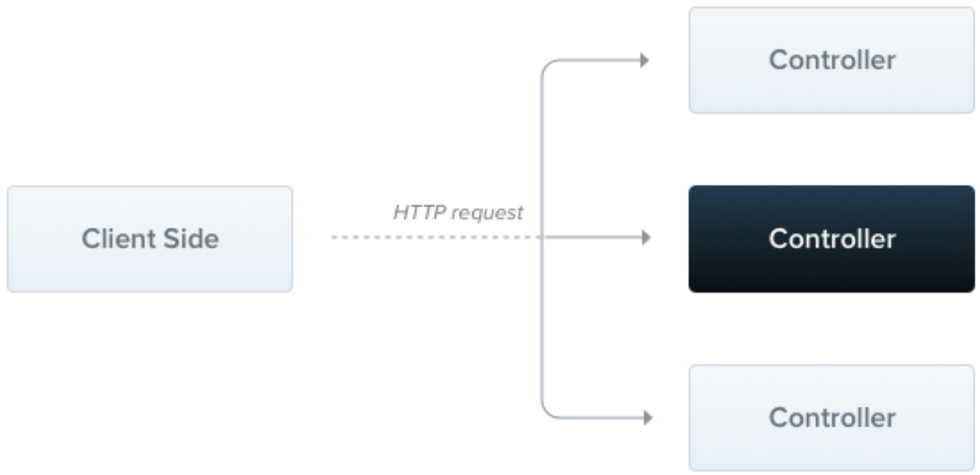
```ts
import { Widget } from '../database/entities/widget.entity';

@Injectable()
export class WidgetsService {
  constructor(
    @Inject('COURSE_REPOSITORY')
    private widgetsRepository: Repository<Widget>
  ) {}

  async getAll(): Promise<Widget[]> {
    return this.widgetsRepository.find();
  }

  async get(id: string): Promise<Widget> {
    const widget = await this.widgetsRepository.findOne(id);
    if (!widget) throw new NotFoundException();
    return widget;
  }
}
```

```ts
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { widgetProviders } from './widget.providers';
import { WidgetsController } from './widgets.controller';
import { WidgetsService } from './widgets.service';

@Module({
  imports: [DatabaseModule],
  controllers: [WidgetsController],
  providers: [...widgetProviders, WidgetsService],
})
export class WidgetsModule {}
```

```ts
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AuthModule } from './auth/auth.module';
import { WidgetsModule } from './widgets/widgets.module';
import { ItemsModule } from './lessons/lessons.module';
import { UsersModule } from './users/users.module';

@Module({
  imports: [
    AuthModule,
    WidgetsModule,
    ItemsModule,
    UsersModule,
  ],
  controllers: [AppController],
  providers: [],
})
export class AppModule {}
```

One more thing...

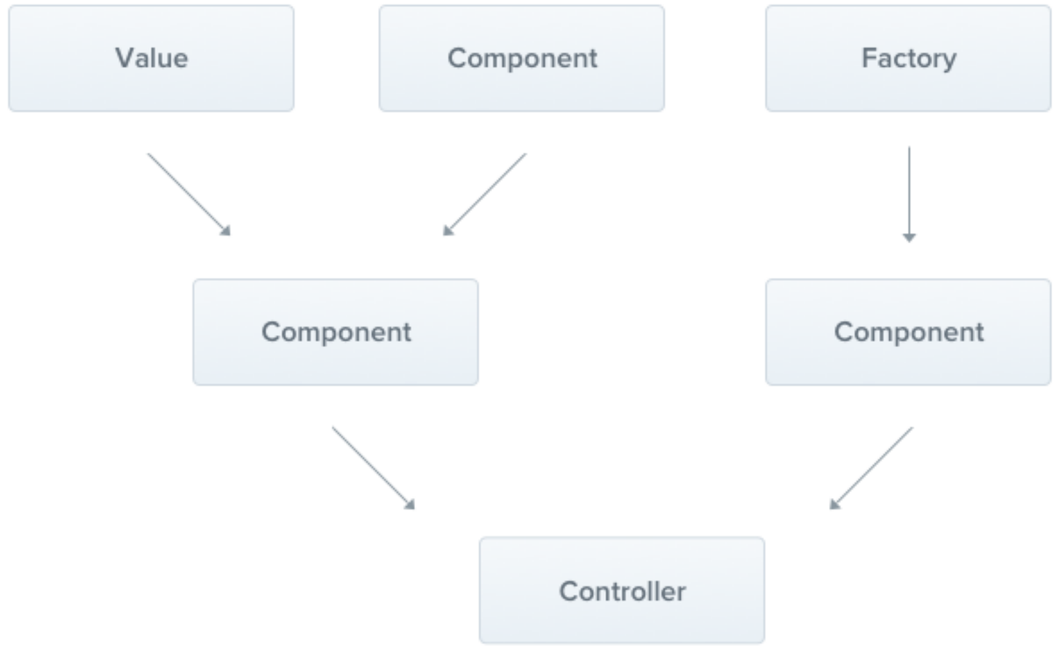
Swagger!
Authentication with Passport
Entities with TypeORM
GraphQL Support
Realtime Data with Socket.IO
Microservice Support
Fastify Support



Controllers

```ts
import { Widget } from '../database/entities/widget.entity';
import { WidgetsService } from './widgets.service';

@Controller('widgets')
export class WidgetsController {
  constructor(private readonly widgetsService: WidgetsService) {}

  @Get()
  all(): Promise<Widget[]> {
    return this.widgetsService.getAll();
  }

  @Get(':id')
  find(@Param() id: string): Promise<Widget> {
    return this.widgetsService.get(id);
  }
}
```

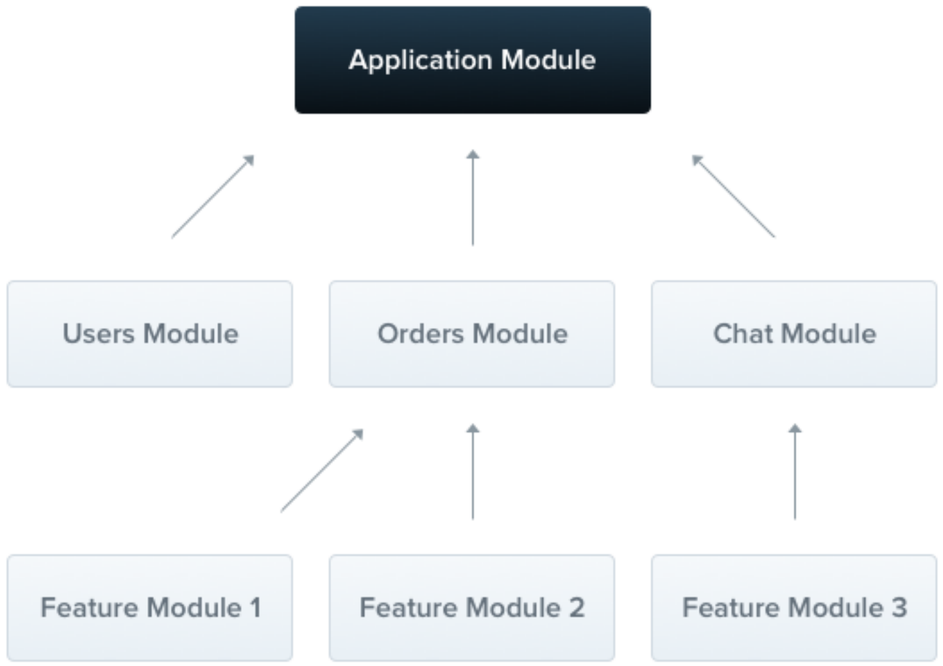
Controllers



Providers

```ts
import { Widget } from '../database/entities/widget.entity';

@Injectable()
export class WidgetsService {
  constructor(
    @Inject('COURSE_REPOSITORY')
    private widgetsRepository: Repository<Widget>
  ) {}

  async getAll(): Promise<Widget[]> {
    return this.widgetsRepository.find();
  }

  async get(id: string): Promise<Widget> {
    const widget = await this.widgetsRepository.findOne(id);
    if (!widget) throw new NotFoundException();
    return widget;
  }
}
```

Services


Application Module
Users Module
Orders Module
Chat Module
Feature Module 1
Feature Module 2
Feature Module 3

Modules

```ts
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { widgetProviders } from './widget.providers';
import { WidgetsController } from './widgets.controller';
import { WidgetsService } from './widgets.service';

@Module({
  imports: [DatabaseModule],
  controllers: [WidgetsController],
  providers: [...widgetProviders, WidgetsService],
})
export class WidgetsModule {}
```

Feature Module

```ts
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AuthModule } from './auth/auth.module';
import { WidgetsModule } from './widgets/widgets.module';
import { ItemsModule } from './lessons/lessons.module';
import { UsersModule } from './users/users.module';

@Module({
  imports: [
    AuthModule,
    WidgetsModule,
    ItemsModule,
    UsersModule,
  ],
  controllers: [AppController],
  providers: [],
})
export class AppModule {}
```

Root Module

The Nest CLI

The Nest CLI

- The Nest CLI is a command-line interface tool that helps you to **initialize**, **develop**, and **maintain** your Nest applications.
- It assists in multiple ways, including **scaffolding** the project, **serving** it in development mode, and **building** and **bundling** the application for production distribution.
- It **embodies best-practice architectural patterns** to encourage well-structured apps.

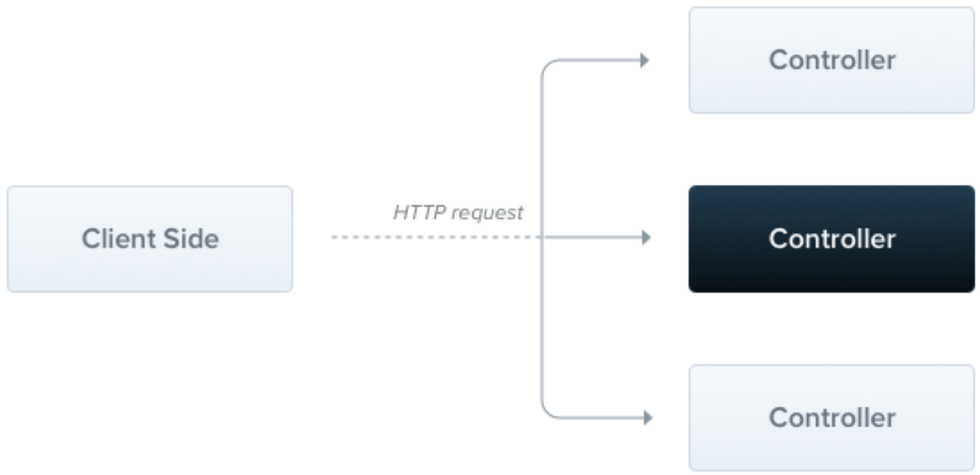
```
$ npm install -g @nestjs/cli

$ nest --help

$ nest generate --help

$ nest new my-nest-project
$ cd my-nest-project
$ npm run start:dev

$ nest generate <schematic> <name> [options]
$ nest g <schematic> <name> [options]
```

Basic Usage

```
$ nx g @nestjs/schematics:resource widgets \
    --type rest \
    --crud true \
    --source-root apps/api/src
```

Create Resource

Nest Controllers



Controllers

```
export class WidgetsController {
  all(): Widget[] {
    return widgets;
  }

  find(id: string): Widget {
    return find(id, widgets);
  }

  create(widget: Widget): Widget {
    widgets = add(widget, widgets);
    return widget;
  }

  update(id: string, widget: Widget): Widget {
    widgets = update(widget, widgets);
    return widget;
  }

  delete(id: string) {
    widgets = remove(id, widgets);
    return {};
  }
}
```

Controllers

```
const find = (id, collection) => collection.find((item) => item.id === id);

const add = (entity, collection) => [...collection, entity];

const update = (entity, collection) =>
  collection.map((item) => (item.id === entity.id ? { ...entity } : item));

const remove = (id, collection) => collection.filter((item) => item.id !== id);
```

Utility Functions

```ts
@Controller('widgets')
export class WidgetsController {
  all(): Widget[] { }

  find(id: string): Widget { }

  create(widget: Widget): Widget { }

  update(id: string, widget: Widget): Widget { }

  delete(@Param('id') id: string) { }
}
```

Expose the Route

```ts
@Controller('widgets')
export class WidgetsController {
  @Get()
  all(): Widget[] { }

  @Get(':id')
  find(id: string): Widget { }

  @Post()
  create(widget: Widget): Widget { }

  @Put(':id')
  update(id: string, widget: Widget): Widget { }

  @Delete(':id')
  delete(@Param('id') id: string) { }
}
```

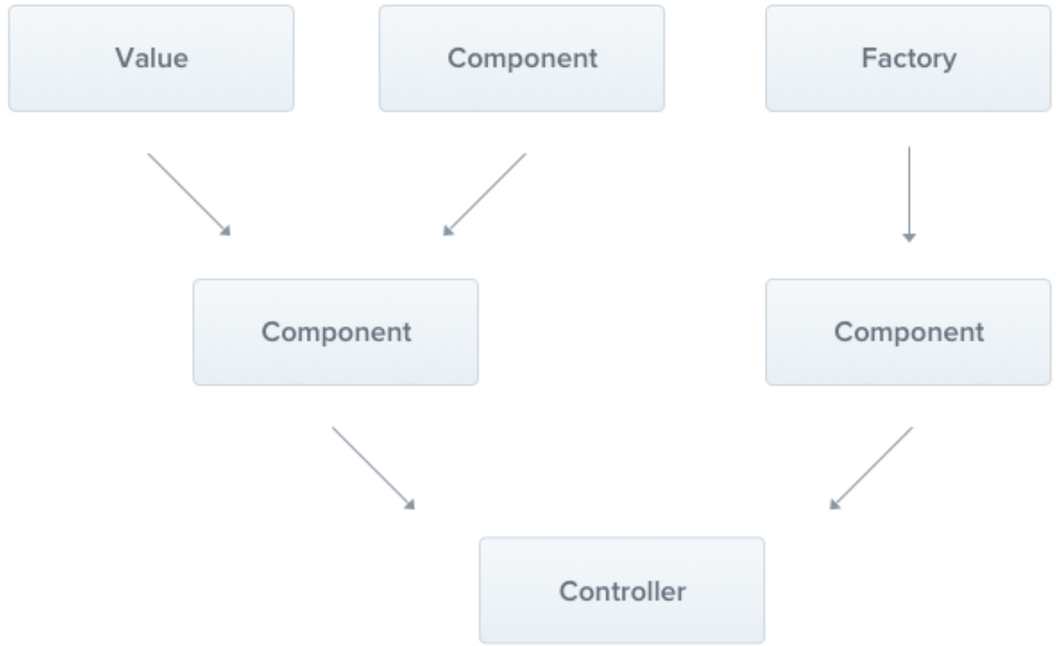
Expose the HTTP Handlers

```ts
@Controller('widgets')
export class WidgetsController {
  @Get()
  all(): Widget[] { }

  @Get(':id')
  find(@Param() id: string): Widget { }

  @Post()
  create(@Body() widget: Widget): Widget { }

  @Put(':id')
  update(@Param('id') id: string, @Body() widget: Widget): Widget { }

  @Delete(':id')
  delete(@Param('id') id: string) { }
}
```

Decorators

```
@Controller('widgets')
export class WidgetsController {
  @Get()
  all(): Widget[] {
    return widgets;
  }

  @Get(':id')
  find(@Param() id: string): Widget {
    return find(id, widgets);
  }

  @Post()
  create(@Body() widget: Widget): Widget {
    widgets = add(widget, widgets);
    return widget;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() widget: Widget): Widget {
    widgets = update(widget, widgets);
    return widget;
  }

  @Delete(':id')
  delete(@Param('id') id: string) {
    widgets = remove(id, widgets);
    return {};
  }
}
```

Nest Providers



Providers

```ts
@Controller('widgets')
export class WidgetsController {
  constructor(private widgetsService: WidgetsService) {}

  @Get()
  all(): Widget[] {
    return this.widgetsService.all();
  }

  @Get(':id')
  find(@Param() id: string): Widget {
    return this.widgetsService.find(id);
  }

  @Post()
  create(@Body() widget: Widget): Widget {
    return this.widgetsService.create(widget);
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() widget: Widget): Widget {
    return this.widgetsService.update(widget);
  }

  @Delete(':id')
  delete(@Param('id') id: string) {
    return this.widgetsService.delete(id);
  }
}
```

```ts
@Controller('widgets')
export class WidgetsController {
  constructor(private widgetsService: WidgetsService) {}

  @Get()
  all(): Widget[] {
    return this.widgetsService.all();
  }

  @Get(':id')
  find(@Param() id: string): Widget {
    return this.widgetsService.find(id);
  }

  @Post()
  create(@Body() widget: Widget): Widget {
    return this.widgetsService.create(widget);
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() widget: Widget): Widget {
    return this.widgetsService.update(widget);
  }

  @Delete(':id')
  delete(@Param('id') id: string) {
    return this.widgetsService.delete(id);
  }
}
```

```ts
@Injectable()
export class WidgetsService {
  all(): Widget[] {
    return widgets;
  }

  find(id: string): Widget {
    return find(id, widgets);
  }

  create(widget: Widget): Widget {
    widgets = add(widget, widgets);
    return widget;
  }

  update(widget: Widget): Widget {
    widgets = update(widget, widgets);
    return widget;
  }

  delete(id: string) {
    widgets = remove(id, widgets);
    return {};
  }
}
```

Using Swagger

Swagger UI

localhost:3333/api/#/default/UsersController_find

Swagger
Supported by SMARTBEAR

NestJS Quickstart API 1.0 OAS3

REST API for the NestJS Quickstart course

default

POST /api/auth/login

GET /api/profile

GET /api/users

POST /api/users

GET /api/users/{id}

Parameters | Cancel

No parameters

Execute | Clear

Responses

Curl

```
curl -X GET "http://localhost:3333/api/users/{id}" -H  "accept: */*"
```

Request URL

```
http://localhost:3333/api/users/{id}
```

Server response

```
$ npm i --save @nestjs/swagger swagger-ui-express
```

Install Dependencies

```
const configureSwagger = (app) => {
  const options = new DocumentBuilder()
    .setTitle('Production Angular API')
    .setDescription('REST API for the Production Angular widget')
    .setVersion('1.0')
    .build();
  const document = SwaggerModule.createDocument(app, options);
  SwaggerModule.setup('api', app, document);
};
```

Configure Swagger

```ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const globalPrefix = 'api';
  const port = process.env.PORT || 3333;

  app.enableCors();
  app.setGlobalPrefix(globalPrefix);

  configureSwagger(app);

  await app.listen(port, () => {
    Logger.log('Listening at http://localhost:' + port + '/' + globalPrefix);
  });
}

bootstrap();
```

Bootstrap Nest

Reactive

Angular

Facade Pattern

And Service with a Subject

Facades

- Facades are **controversial** and can be misused
- Facades are a **pure delegation layer** and should NOT handle business logic
- Facades provide a **clean separation** between components and the rest of your application
- Just as inputs and outputs provide an **API for your components**, Facades provide an **API for your application**
- Facades are an excellent way to **incrementally integrate NgRx**
- Facades are great for **mocking** out a **business logic layer**

We will start with a standard component service combo

```
widgets$: Observable<Widget[]>;
selectedWidget: Widget;

constructor(
  private widgetsService: WidgetsService
) { }

ngOnInit(): void {
  this.loadWidgets();
}

reset() {
  this.loadWidgets();
  this.selectWidget(null);
}

selectWidget(widget: Widget) {
  this.selectedWidget = widget;
}
```

Typical Component

```
loadWidgets() { this.widgets$ = this.widgetsService.all();}

saveWidget(widget: Widget) {
  if (widget.id) {
    this.updateWidget(widget);
  } else {
    this.createWidget(widget);
  }
}

createWidget(widget: Widget) {
  this.widgetsService.create(widget)
    .subscribe(_ => this.reset());
}

updateWidget(widget: Widget) {
  this.widgetsService.update(widget)
    .subscribe(_ => this.reset());
}
```

Typical Component

We will introduce a **facade** to **decouple** the component from the implementation details

```
allWidgets = new Subject<Widget[]>();
selectedWidget = new Subject<Widget>();
mutations = new Subject();
```

```
allWidgets$ = this.allWidgets.asObservable();
selectedWidget$ = this.selectedWidget.asObservable();
mutations$ = this.mutations.asObservable();
```

Exposing State

```
reset() {
  this.mutations.next(true);
}

selectWidget(widget: Widget) {
  this.selectedWidget.next(widget);
}

loadWidgets() {
  this.widgetsService
    .all()
    .subscribe((widgets: Widget[]) => this.allWidgets.next(widgets));
}
```

Updating State

```
saveWidget(widget: Widget) {
  if (widget.id) {
    this.updateWidget(widget);
  } else {
    this.createWidget(widget);
  }
}

createWidget(widget: Widget) {
  this.widgetsService.create(widget).subscribe((_) => this.reset());
}

updateWidget(widget: Widget) {
  this.widgetsService.update(widget).subscribe((_) => this.reset());
}

deleteWidget(widget: Widget) {
  this.widgetsService.delete(widget.id).subscribe((_) => this.reset());
}
```

Remote Server Communication

```
widgets$: Observable<Widget[]> = this.widgetsFacade.allWidgets$;
selectedWidget$: Observable<Widget> = this.widgetsFacade.selectedWidget$;

constructor(
  private widgetsFacade: WidgetsFacade
) { }

ngOnInit(): void {
  this.reset();
  this.widgetsFacade.mutations$.subscribe(_ => this.reset());
}

reset() {
  this.loadWidgets();
  this.selectWidget(null);
}
```

Reactive Component

```
selectWidget(widget: Widget) {
  this.widgetsFacade.selectWidget(widget);
}

loadWidgets() {
  this.widgetsFacade.loadWidgets();
}

saveWidget(widget: Widget) {
  if (widget.id) {
    this.widgetsFacade.updateWidget(widget);
  } else {
    this.widgetsFacade.createWidget(widget);
  }
}

deleteWidget(widget: Widget) {
  this.widgetsFacade.deleteWidget(widget);
}
```

Reactive Component

```
<div class="component-container">
  <div class="list-component">
    <bba-widgets-list [widgets]="widgets$ | async"
      (selected)="selectWidget($event)"
      (deleted)="deleteWidget($event)">
    </bba-widgets-list>
  </div>
  <div class="details-component">
    <bba-widget-details [widget]="selectedWidget$ | async"
      (saved)="saveWidget($event)"
      (cancelled)="reset()">
    </bba-widget-details>
  </div>
</div>
```

Reactive Component

NgRx Tour

BONUS!

redux in 5 minutes


Redux

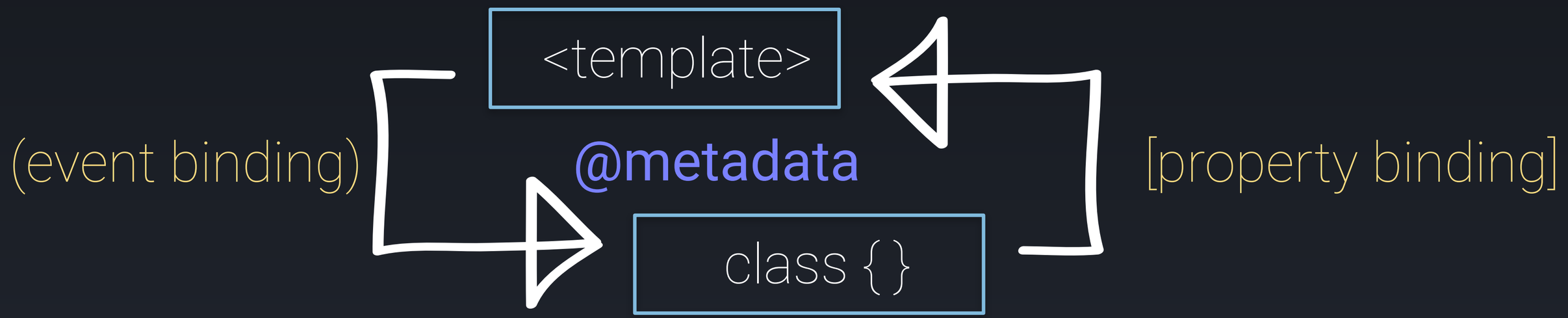
Redux is a library but more importantly it is a **pattern**

Help! I need the 5-minute version!

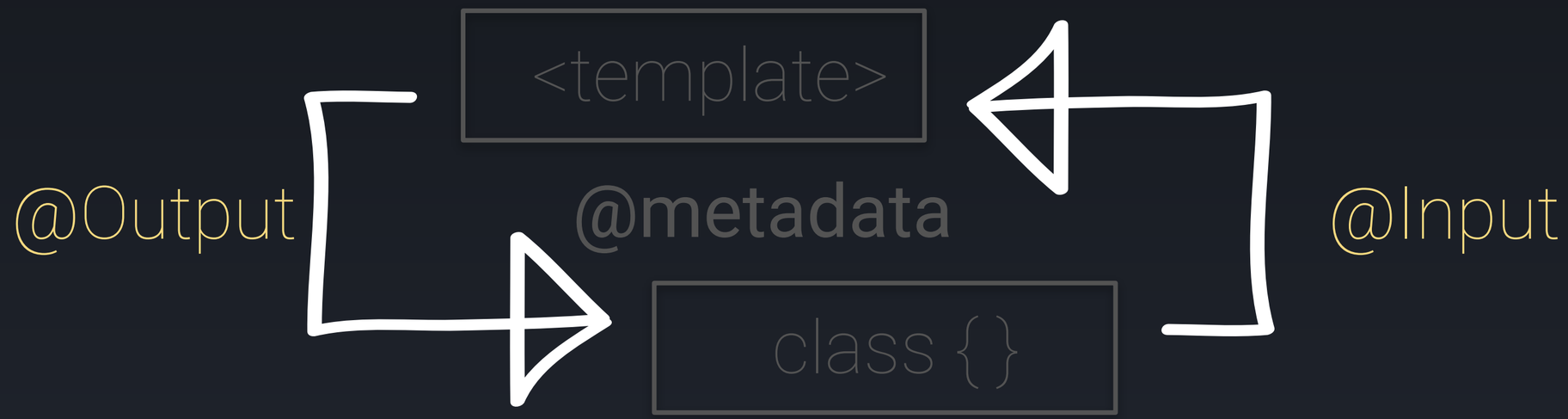
Do you understand

inputs and **outputs**?

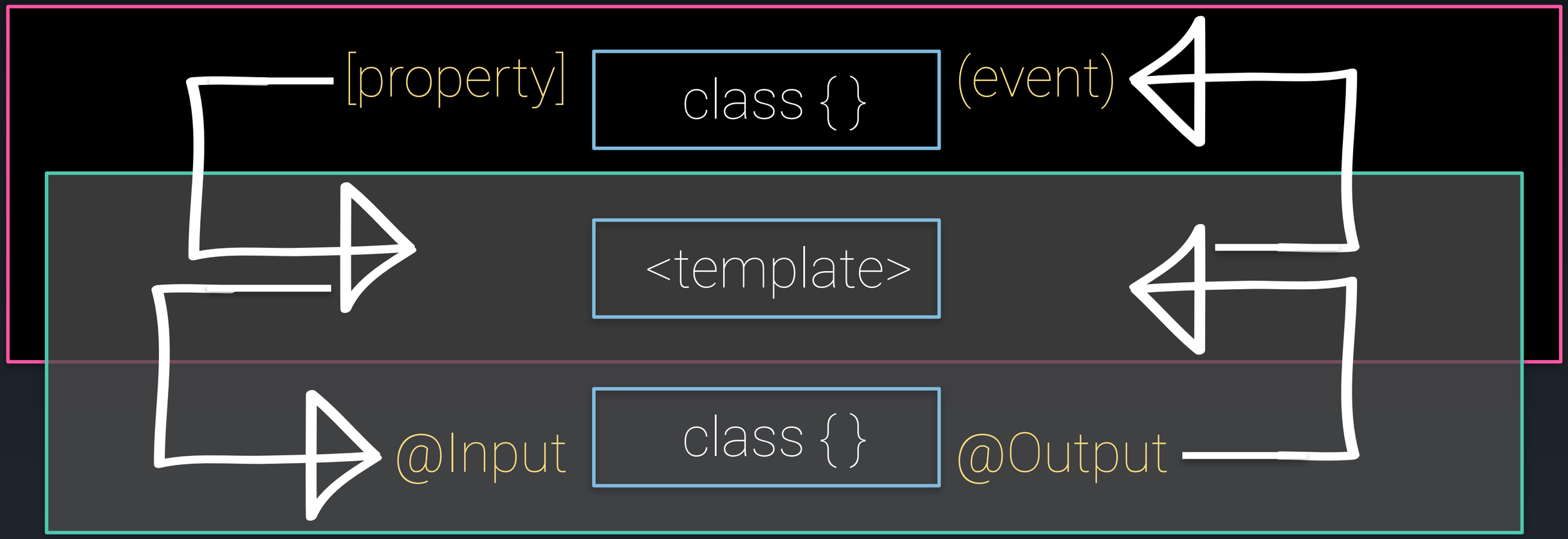
Data Binding



Custom Data Binding

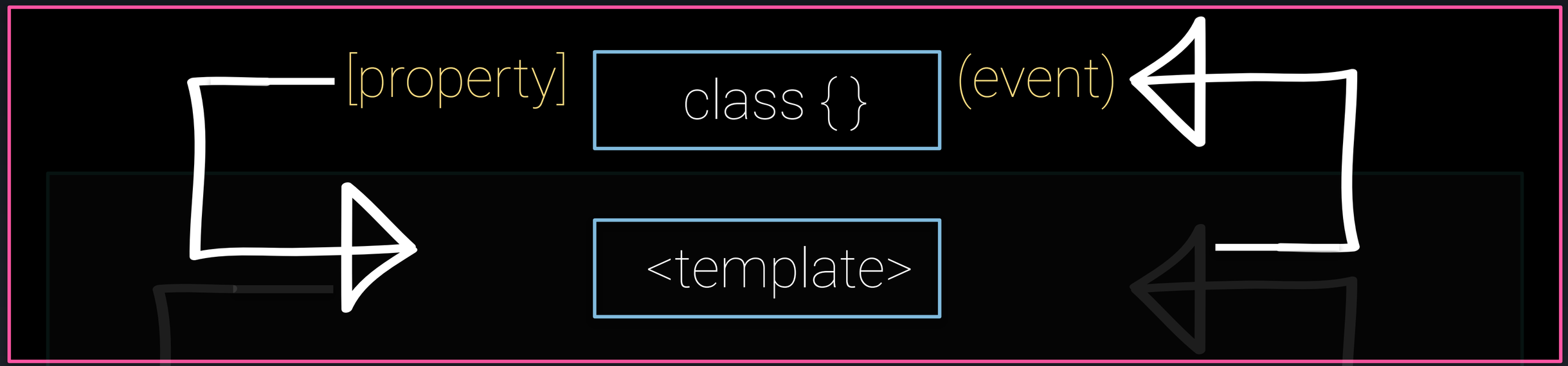


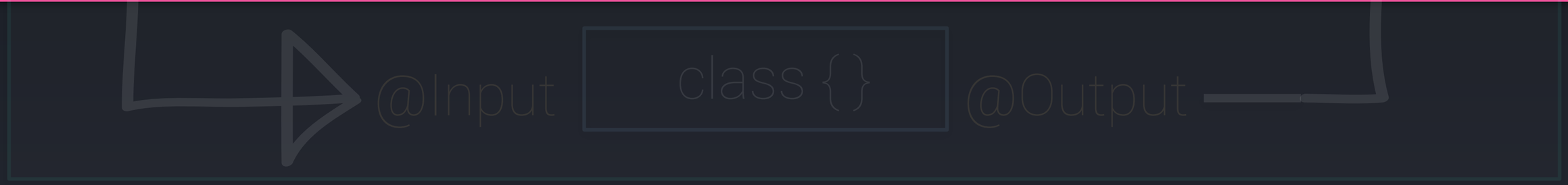
Parent



Child

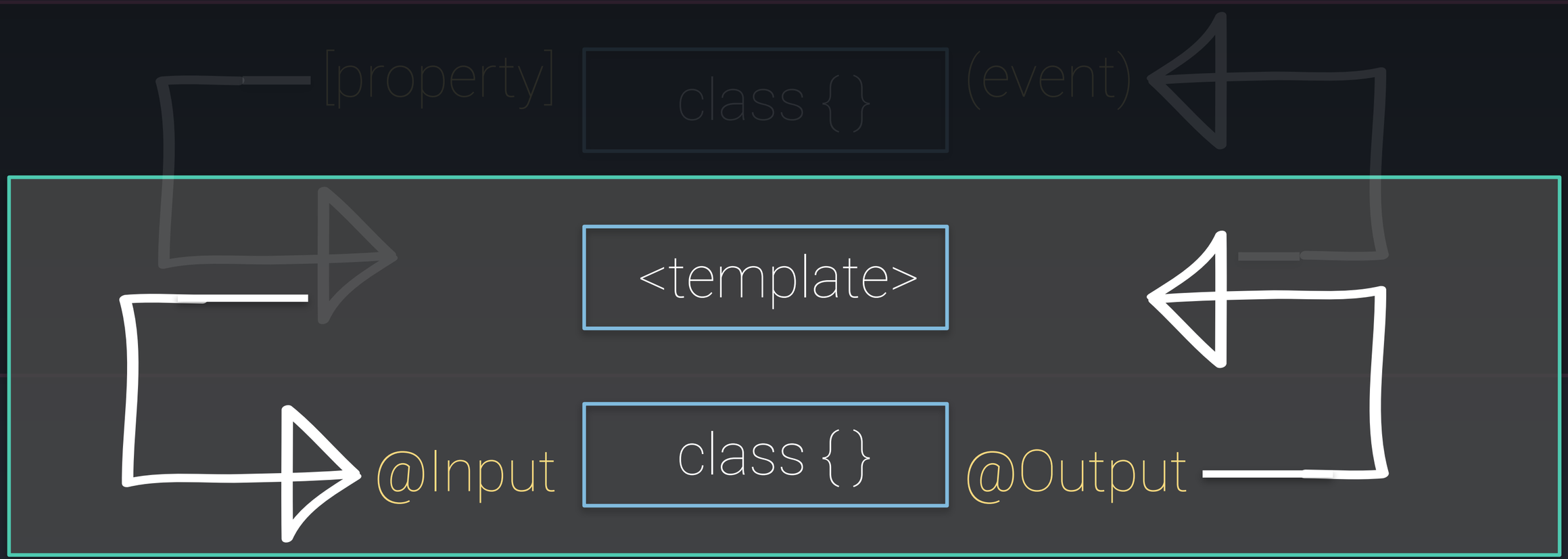
Parent





Child

Parent



Child

Parent

Child

```
<app-items-list
  [items]="items"
  (selected)="selectItem($event)"
  (deleted)="deleteItem($event)">
</app-items-list>
```

Component Contract

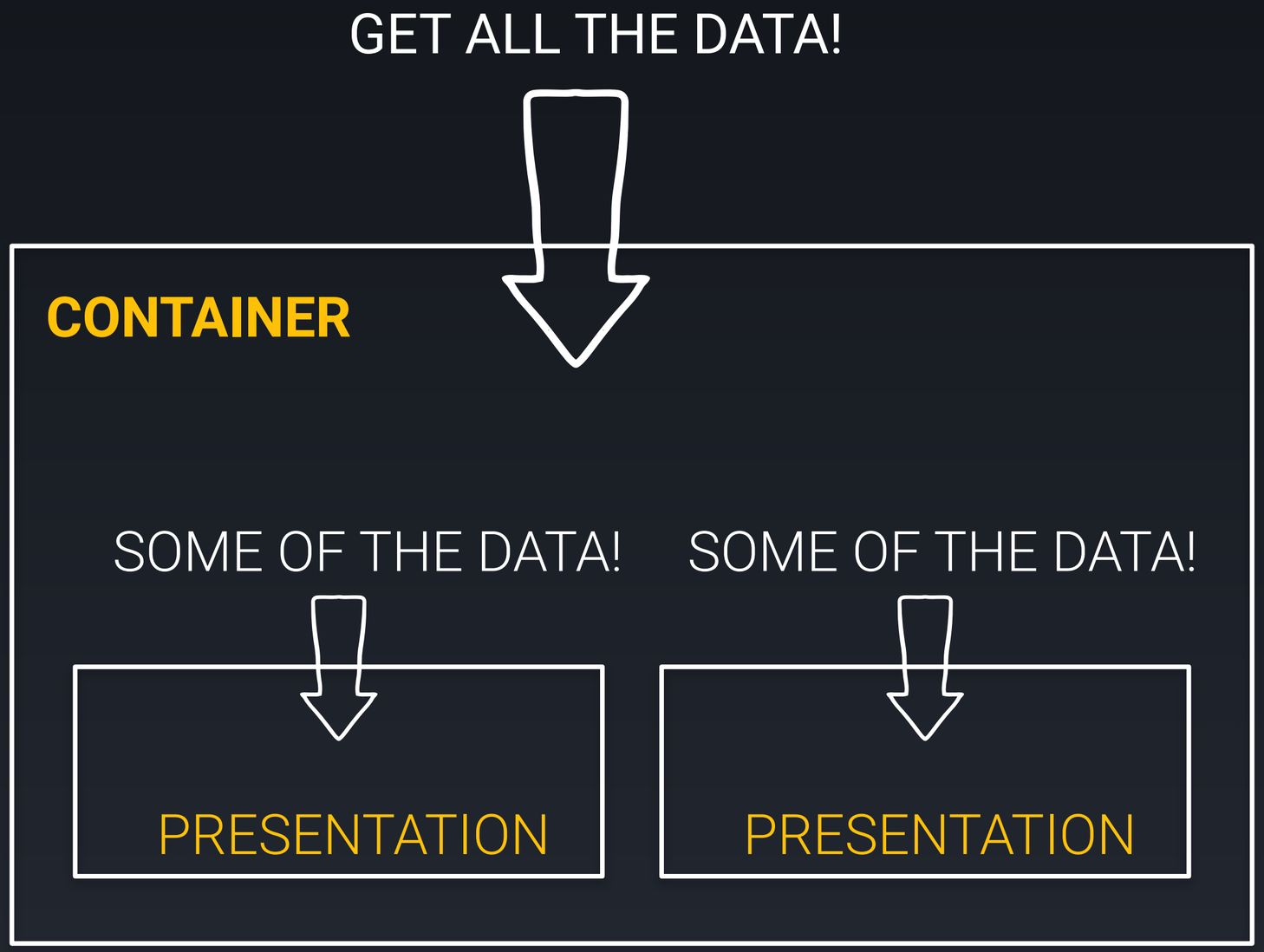
```
<app-items-list
  [items]="items"
  (selected)="selectItem($event)"
  (deleted)="deleteItem($event)">
</app-items-list>
```

Data In

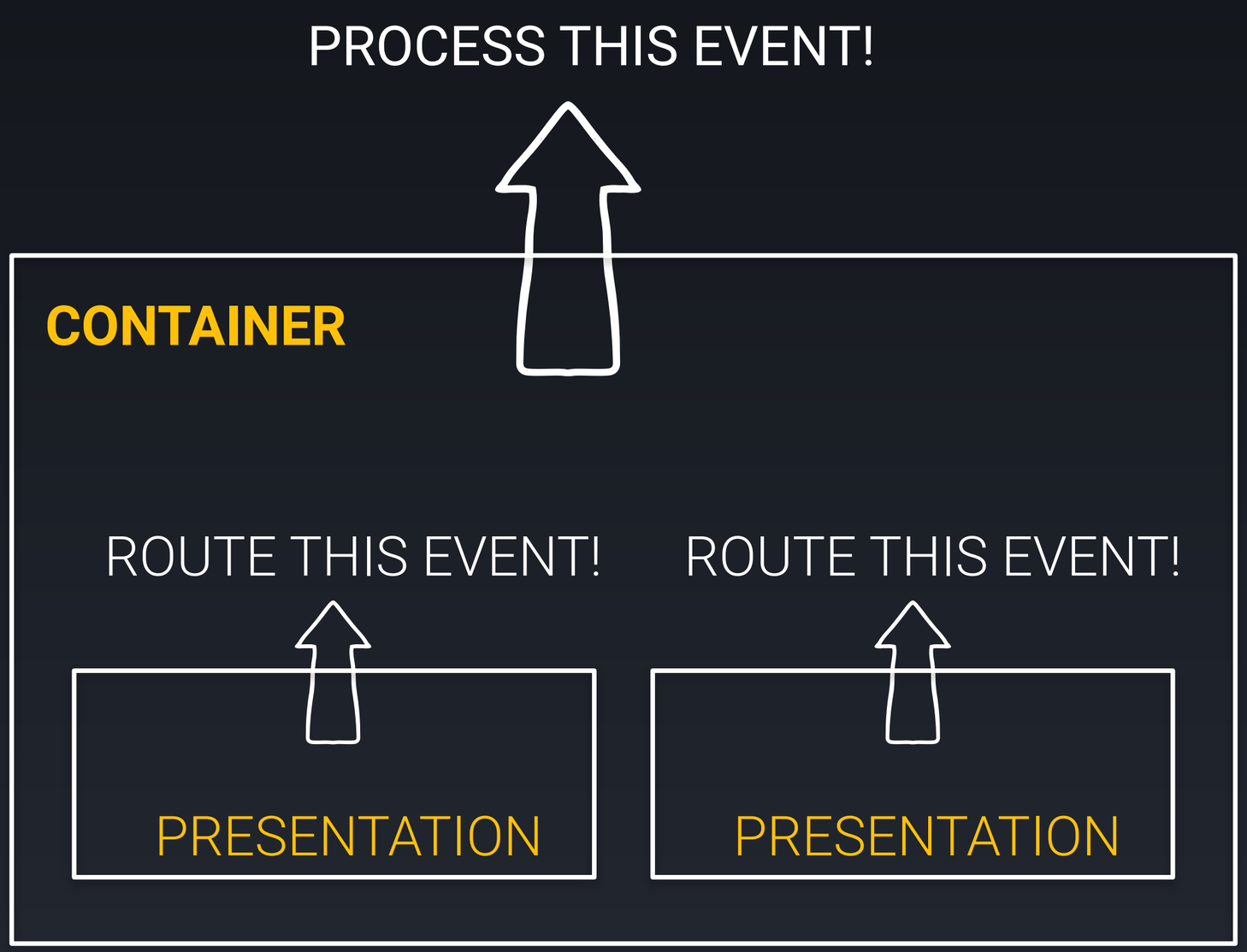
```
<app-items-list
  [items]="items"
  (selected)="selectItem($event)"
  (deleted)="deleteItem($event)">
</app-items-list>
```

Events Out

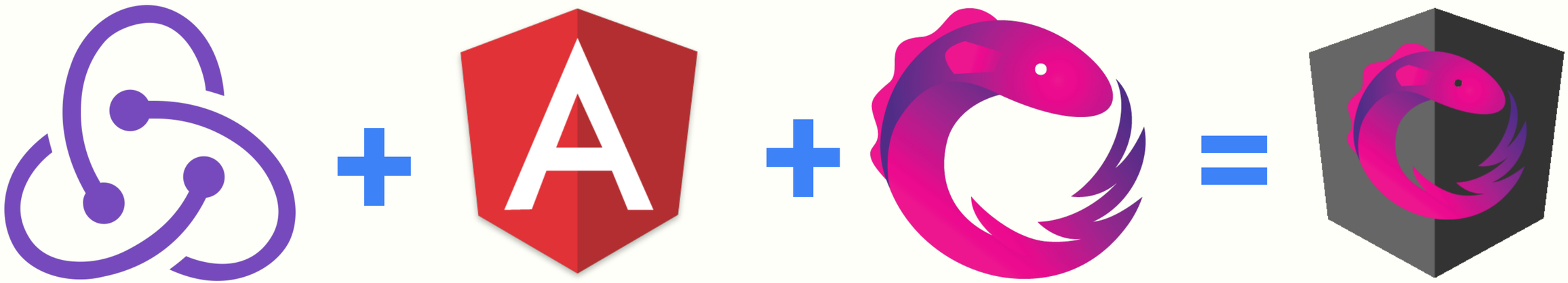
State flows **down**


GET ALL THE DATA!
CONTAINER
SOME OF THE DATA!
SOME OF THE DATA!
PRESENTATION
PRESENTATION

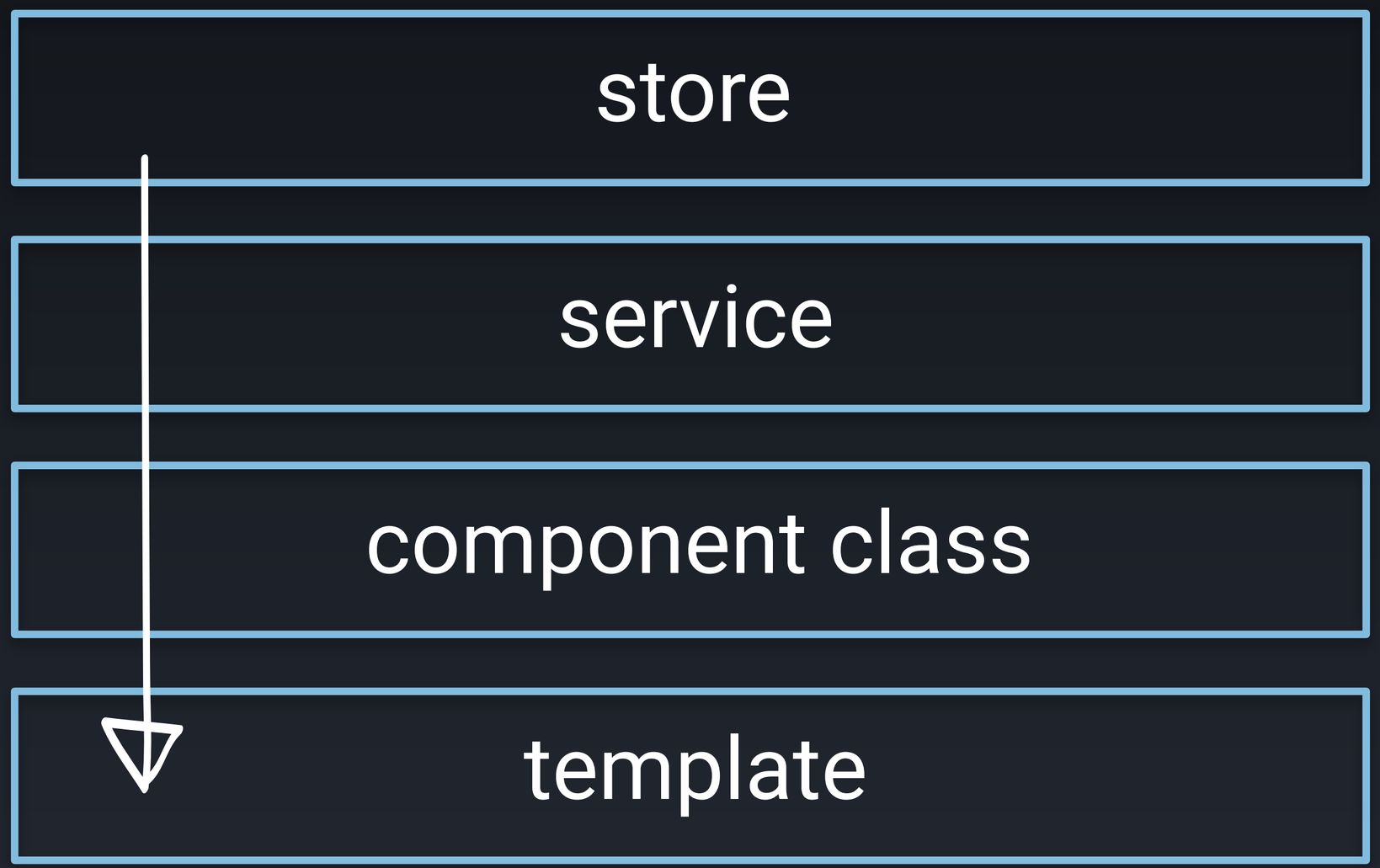
Events flows **up**


PROCESS THIS EVENT!
CONTAINER
ROUTE THIS EVENT!
ROUTE THIS EVENT!
PRESENTATION
PRESENTATION

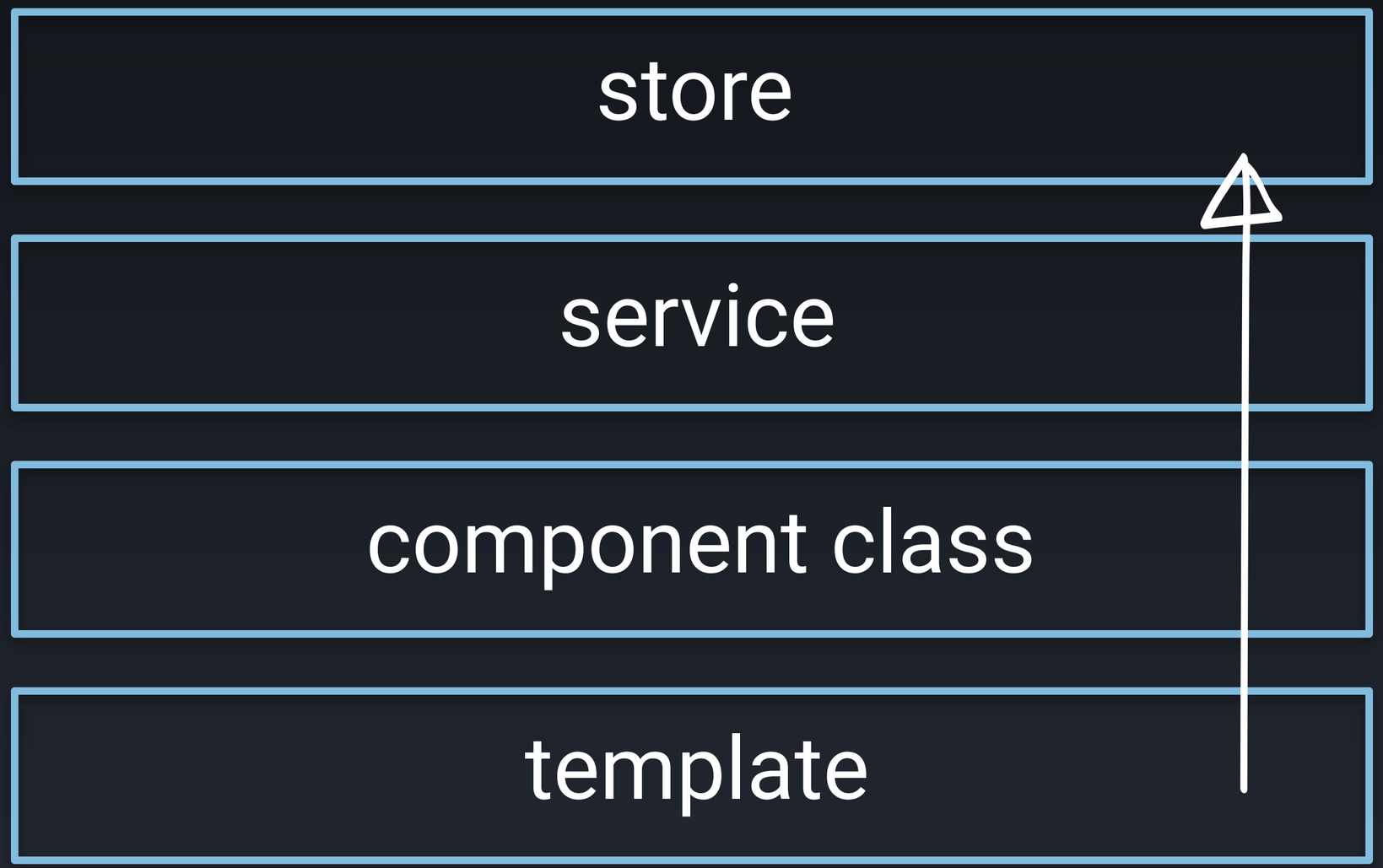
You **basically**
understand redux



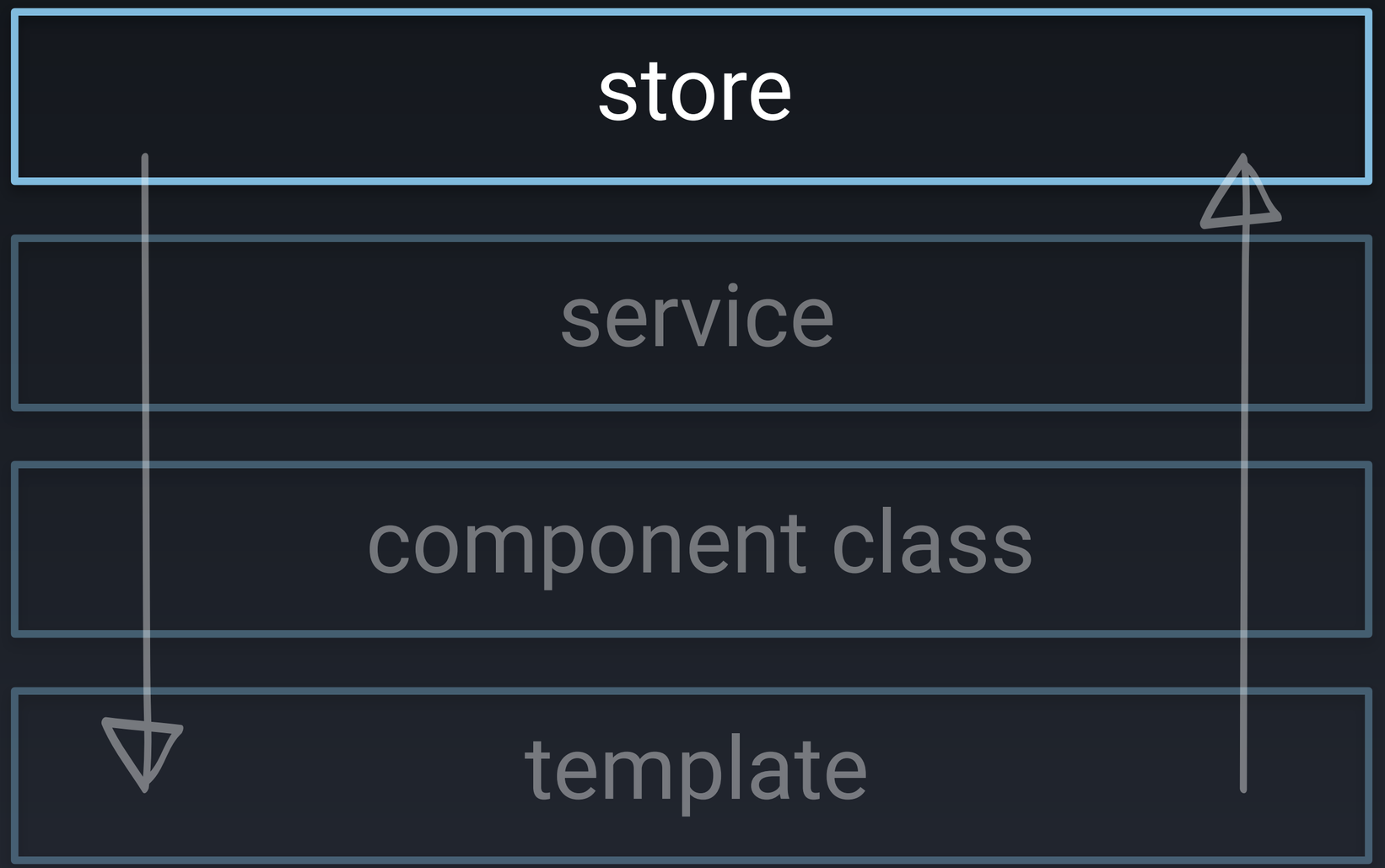
State Flows Down



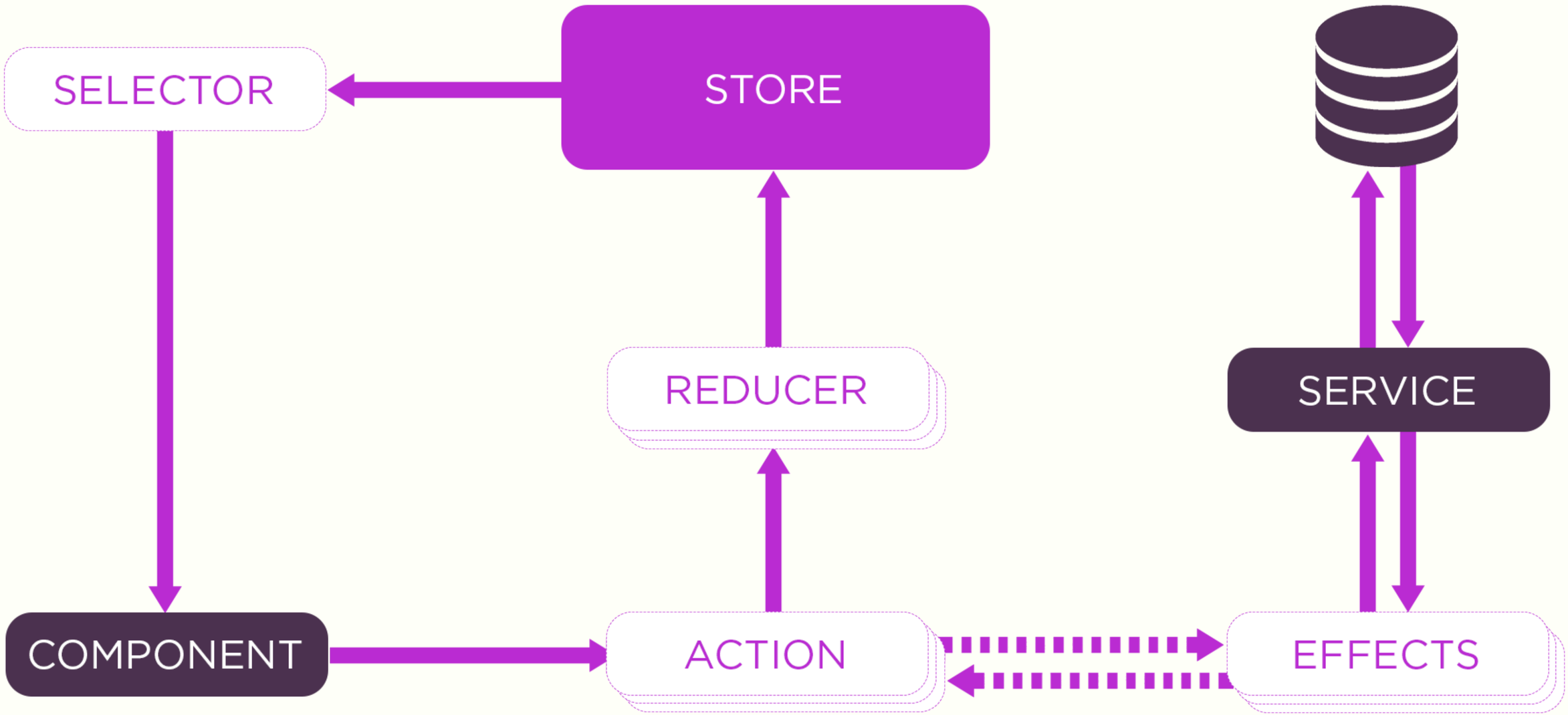
Events Flow Up



Single Source of Truth

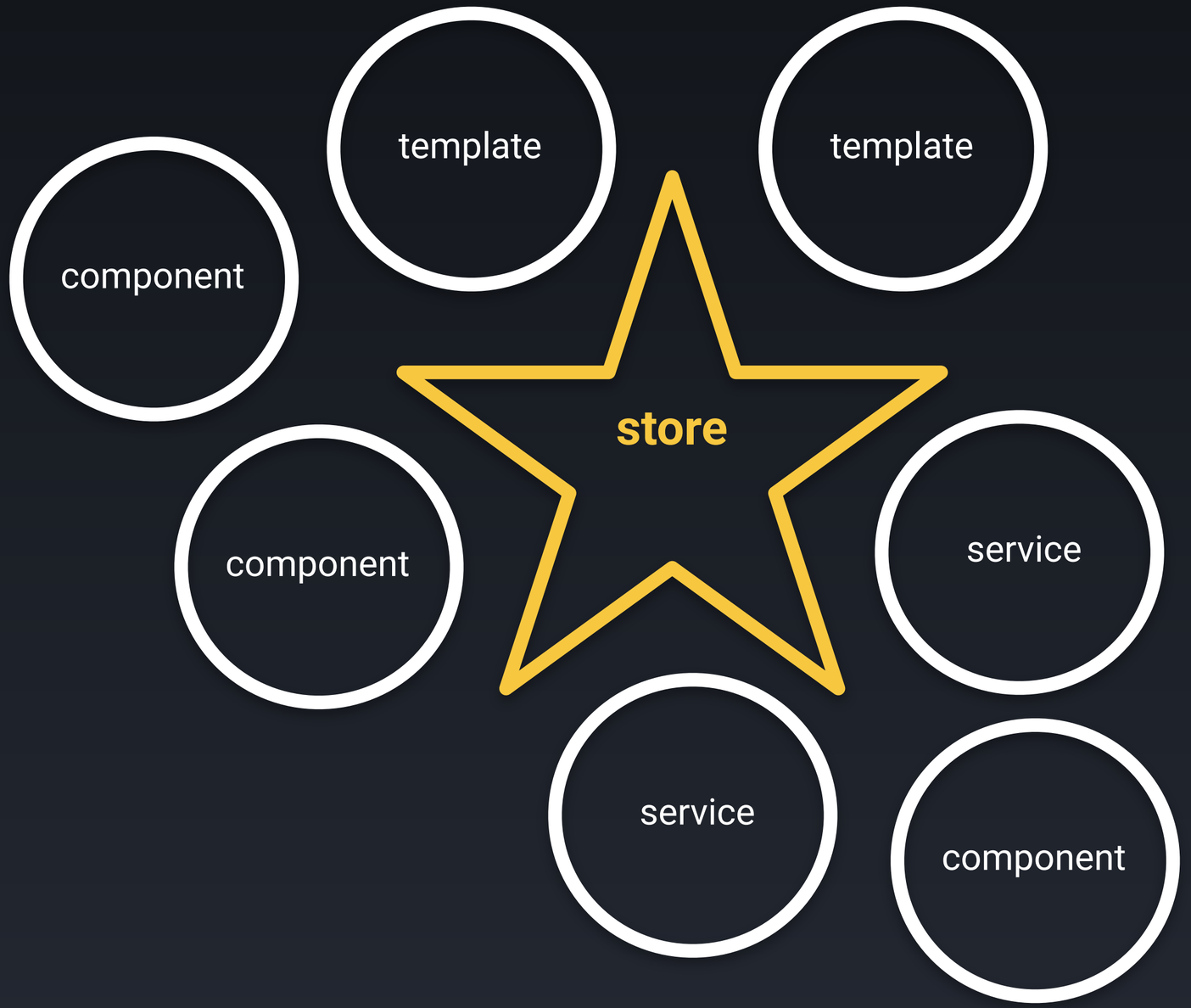


NGRX STATE MANAGEMENT LIFECYCLE



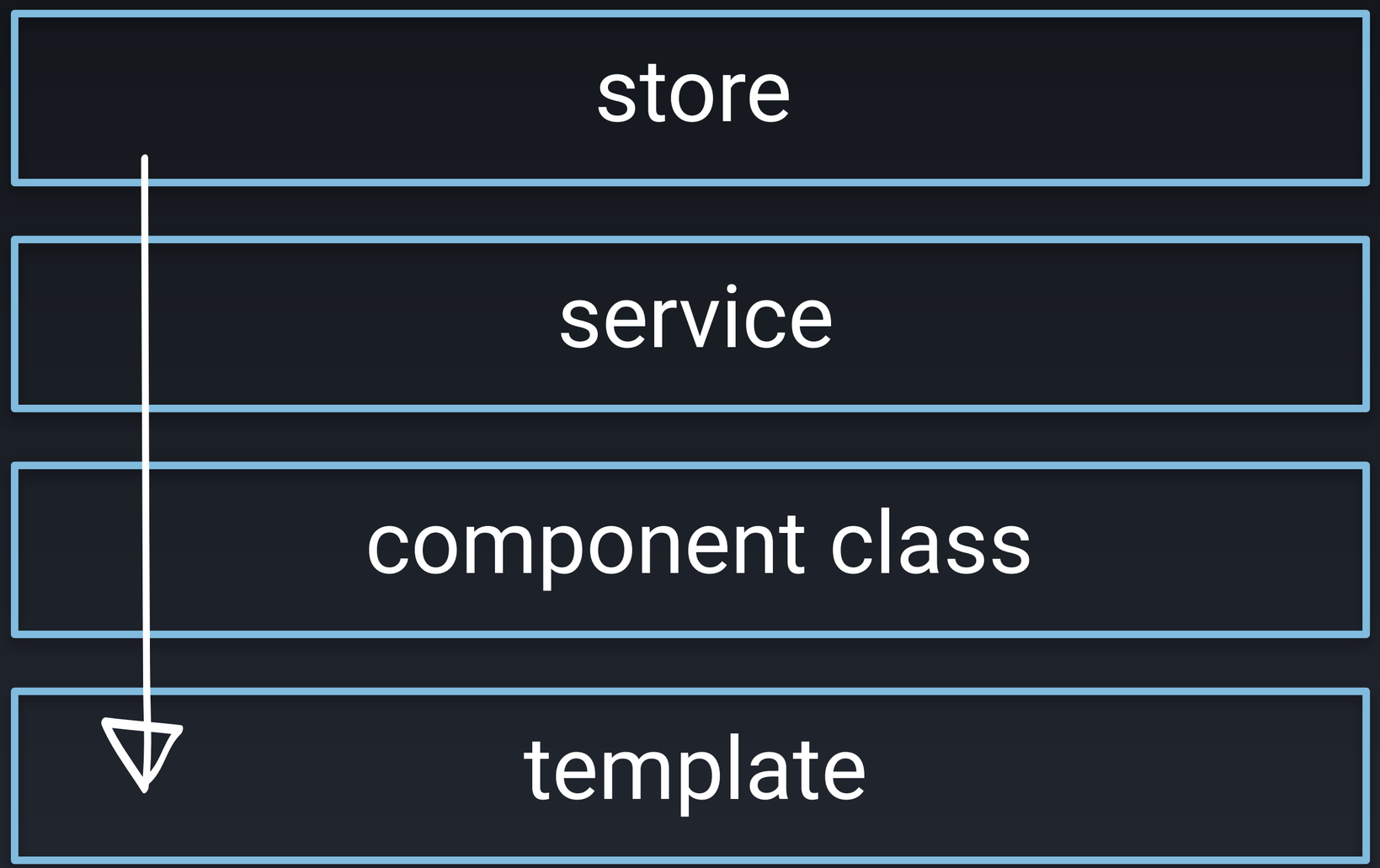
The store is the **single** source of truth

Single State Tree



Application state is accessed via **selectors**

State Flows Down



```ts
export declare class Store<T> extends Observable<T> implements Observer<Action> {
  private actionsObserver;
  private reducerManager;
  constructor(state$: StateObservable, actionsObserver, reducerManager);
  select<K>(mapFn: (state: T) => K): Store<K>;
  lift<R>(operator: Operator<T, R>): Store<R>;
  dispatch<V extends Action = Action>(action: V): void;
  next(action: Action): void;
  error(err: any): void;
  complete(): void;
  addReducer<State, Actions extends Action = Action>(key: string, reducer): void;
  removeReducer<Key extends keyof T>(key: Key): void;
}
```

Store Interface

```ts
export declare class Store<T> extends Observable<T> implements Observer<Action> {
  private actionsObserver;
  private reducerManager;
  constructor(state$: StateObservable, actionsObserver, reducerManager);
  select<K>(mapFn: (state: T) => K): Store<K>;
  lift<R>(operator: Operator<T, R>): Store<R>;
  dispatch<V extends Action = Action>(action: V): void;
  next(action: Action): void;
  error(err: any): void;
  complete(): void;
  addReducer<State, Actions extends Action = Action>(key: string, reducer): void;
  removeReducer<Key extends keyof T>(key: Key): void;
}
```

Concentrated Store Interface

```ts
export declare class State<T> extends BehaviorSubject<any> implements OnDestroy {
  static readonly INIT: "@ngrx/store/init";
  private stateSubscription;
  constructor(
    actions$: ActionsSubject,
    reducer$: ReducerObservable,
    scannedActions: ScannedActionsSubject,
    initialState: any
  );
  ngOnDestroy(): void;
}
```

State Interface

```
export declare class State<T> extends BehaviorSubject<any> implements OnDestroy {
  static readonly INIT: "@ngrx/store/init";
  private stateSubscription;
  constructor(
    actions$: ActionsSubject,
    reducer$: ReducerObservable,
    scannedActions: ScannedActionsSubject,
    initialState: any
  );
  ngOnDestroy(): void;
}
```

Concentrated State Interface

```
// In ItemsFacade
allWidgets$ = this.store.pipe(select(WidgetsSelectors.getAllWidgets));

// In ItemsComponent
widgets$: Observable<Widget[]> = this.widgetsFacade.allWidgets$;
```

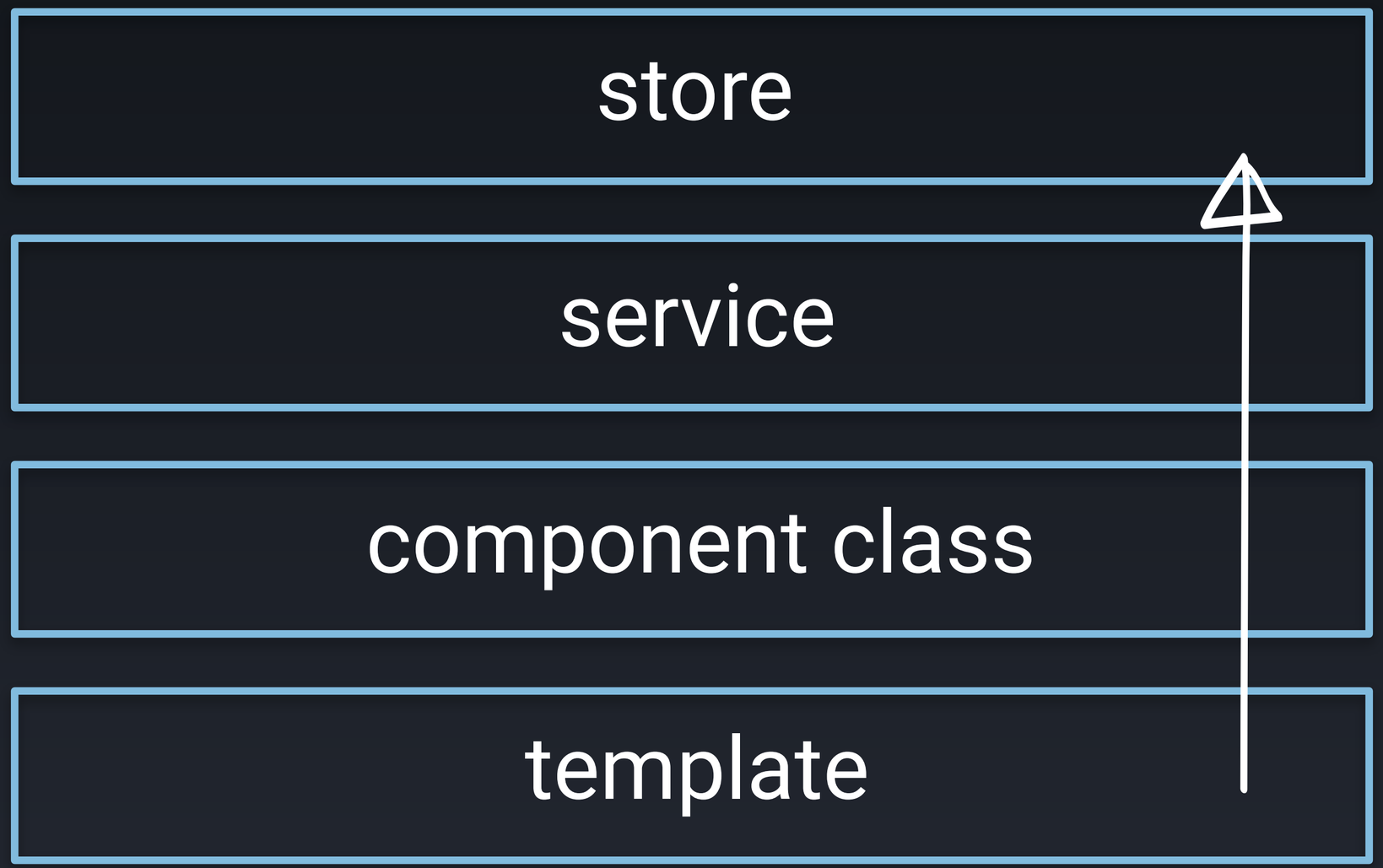
Data Consumption

```
<bba-widgets-list [widgets]="widgets$ | async"
  (selected)="selectWidget($event)"
  (deleted)="deleteWidget($event)">
</bba-widgets-list>
```

Data Consumption

Application events are communicated via **actions**

Events Flow Up



```
loadWidgets() {
  this.widgetsFacade.loadWidgets();
}

selectWidget(widget: Widget) {
  this.widgetsFacade.selectWidget(widget.id);
}

saveWidget(widget: Widget) {
  if (widget.id) {
    this.widgetsFacade.updateWidget(widget);
  } else {
    this.widgetsFacade.createWidget(widget);
  }
}

deleteWidget(widget: Widget) {
  this.widgetsFacade.deleteWidget(widget);
}
```

Event Delegation in Component

```
loadWidgets() {
  this.dispatch(WidgetsActions.loadWidgets());
}

createWidget(widget: Widget) {
  this.dispatch(WidgetsActions.createWidget({ widget }));
}

updateWidget(widget: Widget) {
  this.dispatch(WidgetsActions.updateWidget({ widget }));
}

deleteWidget(widget: Widget) {
  this.dispatch(WidgetsActions.deleteWidget({ widget }));
}

dispatch(action: Action) {
  this.store.dispatch(action);
}
```

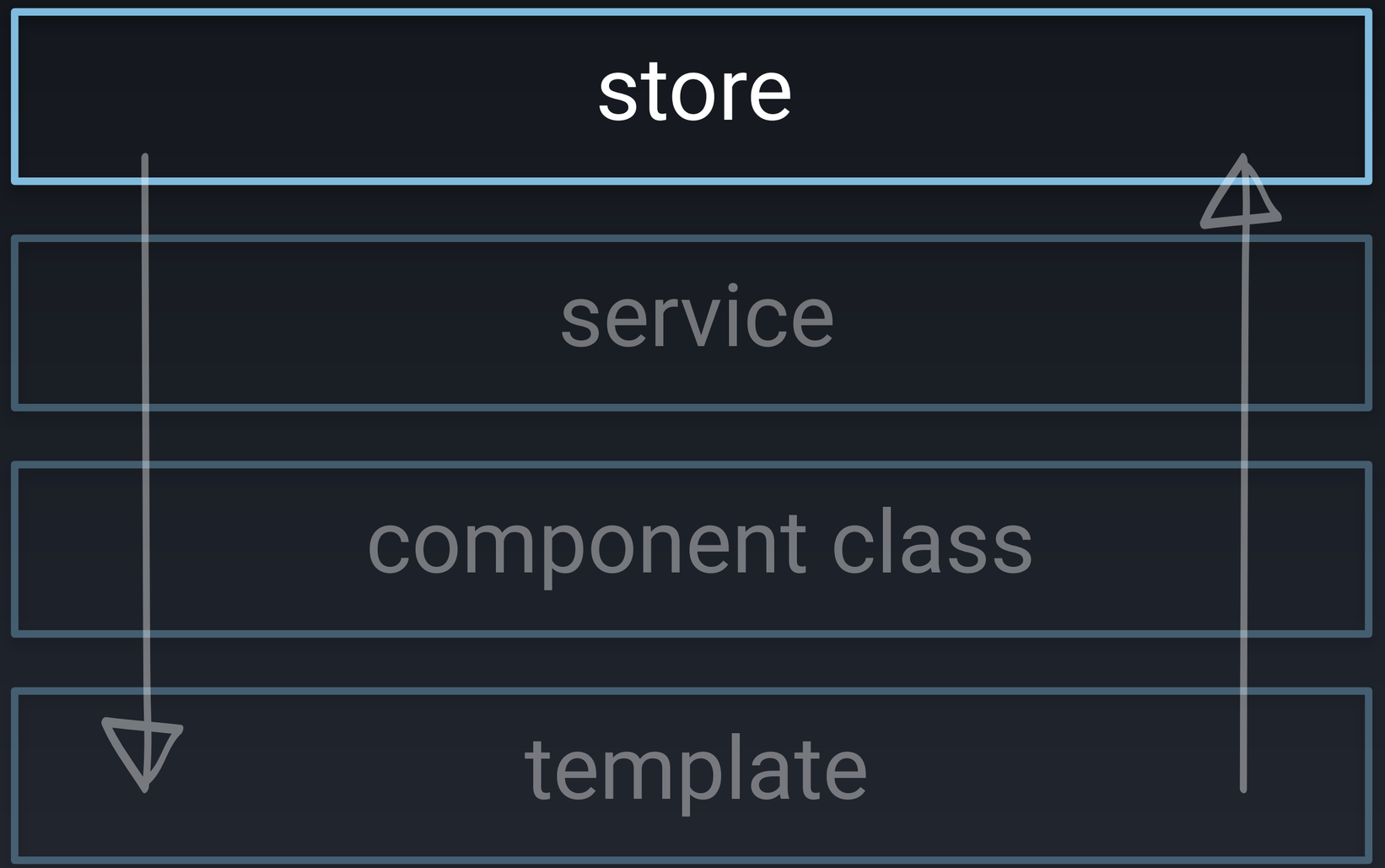
Event Delegation in Facade

```ts
export const loadWidgets = createAction('[Widgets] Load Widgets');

export const createWidget = createAction(
  '[Widgets] Create Widget',
  props<{ widget: Widget }>()
);

export const updateWidget = createAction(
  '[Widgets] Update Widget',
  props<{ widget: Widget }>()
);

export const deleteWidget = createAction(
  '[Widgets] Delete Widget',
  props<{ widget: Widget }>()
);
```

Event Conversion Into Actions

Application state is modified through **reducers**

Single Source of Truth



```
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(
    WidgetsActions.loadWidgetsSuccess, (state, { widgets }) =>
      widgetsAdapter.setAll(widgets, { ...state, loaded: true })
  ),
  on(WidgetsActions.createWidgetSuccess, (state, { widget }) =>
      widgetsAdapter.addOne(widget, state)
  ),
  on(WidgetsActions.updateWidgetSuccess, (state, { widget }) =>
      widgetsAdapter.updateOne({ id: widget.id, changes: widget }, state)
  ),
  on(WidgetsActions.deleteWidgetSuccess, (state, { widget }) =>
      widgetsAdapter.removeOne(widget.id, state)
  )
);
```

State Mutation in Reducer

```ts
export interface AppState {
  router: fromRouter.RouterReducerState<RouterStateUrl>;
  [fromWidgets.COURSES_FEATURE_KEY]: fromWidgets.WidgetsState;
  [fromItems.LESSONS_FEATURE_KEY]: fromItems.ItemsState;
  [fromUsers.USERS_FEATURE_KEY]: fromUsers.UsersState;
}

export const reducers: ActionReducerMap<AppState> = {
  router: fromRouter.routerReducer,
  [fromWidgets.COURSES_FEATURE_KEY]: fromWidgets.widgetsReducer,
  [fromItems.LESSONS_FEATURE_KEY]: fromItems.lessonsReducer,
  [fromUsers.USERS_FEATURE_KEY]: fromUsers.usersReducer,
};
```

Combined Reducers

```ts
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forRoot(reducers, storeConfig),
    EffectsModule.forRoot([WidgetsEffects, ItemsEffects, UsersEffects]),
    StoreDevtoolsModule.instrument({ maxAge: 25, name: STORE_NAME }),
    StoreRouterConnectingModule.forRoot({ stateKey: 'router' }),
  ],
})
export class CoreStateModule {}
```

StoreModule.forRoot

```
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forRoot(reducers, storeConfig),
    EffectsModule.forRoot([WidgetsEffects, ItemsEffects, UsersEffects]),
    StoreDevtoolsModule.instrument({ maxAge: 25, name: STORE_NAME }),
    StoreRouterConnectingModule.forRoot({ stateKey: 'router' }),
  ],
})
export class CoreStateModule {}
```

StoreModule.forRoot

Setting Up The **Store**

The Store

- The store is a **single state tree** that encompasses the **entire application state**
- As far as we are concerned, the store is **read only**
- We work with “slices” of the store via **reducers**
- All state mutations happen via a **reducer**
- It is helpful to think of the store as the **database** of your application and reducers as **tables**

```
const initialWidgets = [
  {
    id: '1',
    title: 'Initial Widget',
    description: 'Pending...'
  }
];
```

```
export interface WidgetsState {
  selectedId: string | null;
  widgets: Widget[];
}

export const initialWidgetsState: WidgetsState = {
  selectedId: null,
  widgets: initialWidgets
}
```

The State Shape

```
const initialWidgets = [
  {
    id: '1',
    title: 'Initial Widget',
    description: 'Pending...'
  }
];
```

```
export interface WidgetsState {
  selectedId: string | null;
  widgets: Widget[];
}

export const initialWidgetsState: WidgetsState = {
  selectedId: null,
  widgets: initialWidgets
}
```

Initial State

```tsx
export function widgetsReducer(
  state: WidgetsState = initialWidgetsState,
  action: Action
  ) {
  switch(action.type) {
    default:
      return state;
  }
}
```

Basic Reducer

```ts
export interface AppState {
  router: fromRouter.RouterReducerState<RouterStateUrl>;
  [fromWidgets.COURSES_FEATURE_KEY]: fromWidgets.WidgetsState;
}

export const reducers: ActionReducerMap<AppState> = {
  router: fromRouter.routerReducer,
  [fromWidgets.COURSES_FEATURE_KEY]: fromWidgets.widgetsReducer,
};
```

Combined Reducers

```ts
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forRoot(reducers, storeConfig),
    StoreDevtoolsModule.instrument({ maxAge: 25, name: STORE_NAME }),
    StoreRouterConnectingModule.forRoot({ stateKey: 'router' }),
  ],
})
export class CoreStateModule {}
```

StoreModule.forRoot

```
export class WidgetsFacade {
  allWidgets$ = this.store.pipe(
    select('widgets'),
    map(state => state.widgets)
  );

  constructor(
    private store: Store<{ widgets: fromWidgets.WidgetsState}>
  ) { }
}
```

store.select

```ts
export const COURSES_FEATURE_KEY = 'widgets';

export interface WidgetsPartialState {
  readonly [COURSES_FEATURE_KEY]: WidgetsState;
}
```

Partial State

```ts
export class WidgetsFacade {
  allWidgets$ = this.store.pipe(
    select('widgets'),
    map(state => state.widgets)
  );

  constructor(
    private store: Store<fromWidgets.WidgetsPartialState>
  ) { }
}
```

Store with Partial State

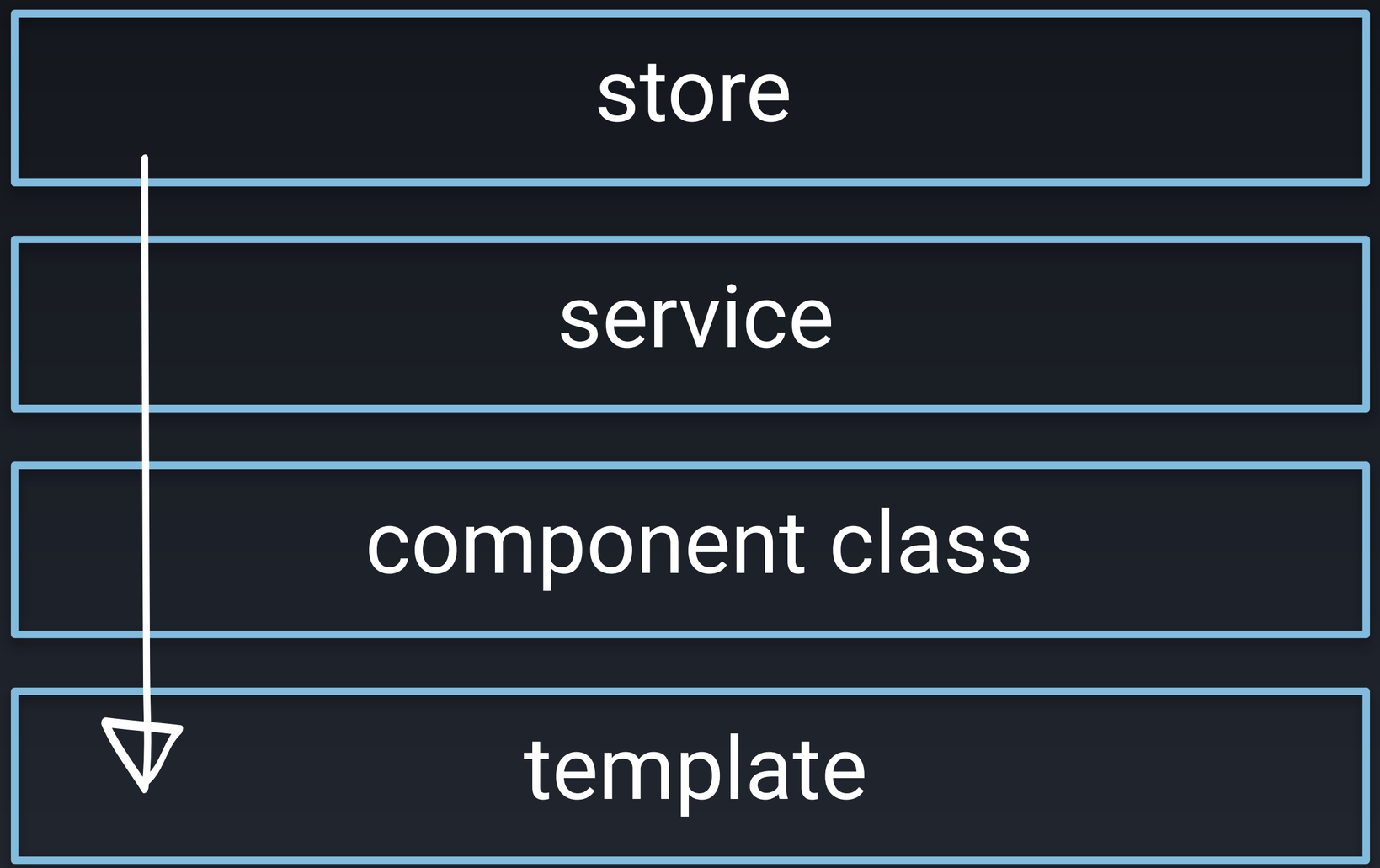
```ts
export class WidgetsComponent implements OnInit {
  widgets$: Observable<Widget[]> = this.widgetsFacade.allWidgets$;

  constructor(
    private widgetsFacade: WidgetsFacade
  ) { }

  ngOnInit(): void { }
}
```

Component Consumption

```
<bba-widgets-list [widgets]="widgets$ | async"
  (selected)="selectWidget($event)"
  (deleted)="deleteWidget($event)">
</bba-widgets-list>
```

Template Consumption

State Flows Down



Managing State with Reducers

Reducers

- All state mutation is handled in a **reducer**
- A reducer has a parameter signature that always includes **initial state** or **current state** and the **current action**
- An **action** object always has a **type** property and additional dynamic properties
- The reducer contains functions to **handle state changes** for the associated action
- Reducers always update application state using pure, **immutable operations**

```
export function widgetsReducer(
  state: WidgetsState = initialWidgetsState,
  action: Action
  ) {
  switch(action.type) {
    case 'selectWidget':
      return {
        selectedId: action['selectedId'],
        widgets: state.widgets
      }
    case 'setAllWidgets':
      return { selectedId: state.selectedId, widgets: action['widgets']}
    case 'createWidget':
      return { selectedId: state.selectedId, widgets: create(state.widgets, action['widget'])}
    case 'updateWidget':
      return { selectedId: state.selectedId, widgets: update(state.widgets, action['widget'])}
    case ‘removeWidget':
      return { selectedId: state.selectedId, widgets: remove(state.widgets, action['widget'])}
    default:
      return state;
  }
}
```

Reducer

```
export function widgetsReducer(
  state: WidgetsState = initialWidgetsState,
  action: Action
  ) {
  switch(action.type) {
    case 'selectWidget':
      return {
        selectedId: action['selectedId'],
        widgets: state.widgets
      }
    case 'setAllWidgets':
      return { selectedId: state.selectedId, widgets: action['widgets']}
    case 'createWidget':
      return { selectedId: state.selectedId, widgets: create(state.widgets, action['widget'])}
    case 'updateWidget':
      return { selectedId: state.selectedId, widgets: update(state.widgets, action['widget'])}
    case ‘removeWidget':
      return { selectedId: state.selectedId, widgets: remove(state.widgets, action['widget'])}
    default:
      return state;
  }
}
```

Action Types

```
export function widgetsReducer(
  state: WidgetsState = initialWidgetsState,
  action: Action
  ) {
  switch(action.type) {
    case 'selectWidget':
      return {
        selectedId: action['selectedId'],
        widgets: state.widgets
      }
    case 'setAllWidgets':
      return { selectedId: state.selectedId, widgets: action['widgets']}
    case 'createWidget':
      return { selectedId: state.selectedId, widgets: create(state.widgets, action['widget'])}
    case 'updateWidget':
      return { selectedId: state.selectedId, widgets: update(state.widgets, action['widget'])}
    case ‘removeWidget':
      return { selectedId: state.selectedId, widgets: remove(state.widgets, action['widget'])}
    default:
      return state;
  }
}
```

Action Handlers

```
const create = (collection, obj) => [...collection, obj];
const update = (collection, obj) => collection.map(i => {
  return i.id === obj.id ? Object.assign({}, obj) : i;
});
const remove = (collection, obj) => collection.filter(i => i.id === obj.id);
```

Immutable Operations

```
createWidget(widget: Widget) {
  this.store.dispatch({ type: 'createWidget', widget });
}

updateWidget(widget: Widget) {
  this.store.dispatch({ type: 'updateWidget', widget });
}

deleteWidget(widget: Widget) {
  this.store.dispatch({ type: ‘deleteWidget', widget });
}
```

store.dispatch

```
loadWidgets() {
  this.widgetsService
    .all()
    .subscribe((widgets: Widget[]) =>
      this.store.dispatch({ type: 'setAllWidgets', widgets })
    );
}
```

store.dispatch

```
loadWidgets() {
  this.widgetsService
    .all()
    .subscribe((widgets: Widget[]) =>
      this.store.dispatch({ type: 'setAllWidgets', widgets })
    );
}
```

store.dispatch

Event Communication with **Actions**

Actions

- **Events** are communicated via **action** objects
- With TypeScript, we can leverage **strongly typed actions** to prevent **naming collisions**
- Before **action creators**, you would create actions using **class-based action creators**
- The **createAction** function returns a function that will return an **Action** compliant object when called
- The **props** method defines additional metadata for that action

Actions Best Practices

- **Upfront** - write actions before developing features to understand and gain a shared knowledge of the feature being implemented.
- **Divide** - categorize actions based on the event source.
- **Many** - actions are inexpensive to write, so the more actions you write, the better you express flows in your application.
- **Event-Driven** - capture events *not commands* as you are separating the description of an event and the handling of that event.
- **Descriptive** - provide context that are targeted to a unique event with more detailed information you can use to aid in debugging with the developer tools.

```
createWidget(widget: Widget) {
  this.store.dispatch({ type: 'create', widget });
}

updateWidget(widget: Widget) {
  this.store.dispatch({ type: 'update', widget });
}

deleteWidget(widget: Widget) {
  this.store.dispatch({ type: 'delete', widget });
}
```

Weak Action Types

```
export function widgetsReducer(
  state: WidgetsState = initialWidgetsState,
  action: Action
  ) {
  switch(action.type) {
    case 'select':
      return {
        selectedId: action['selectedId'],
        widgets: state.widgets
      }
    case 'setAll':
      return { selectedId: state.selectedId, widgets: action['widgets']}
    case 'create':
      return { selectedId: state.selectedId, widgets: create(state.widgets, action['widget'])}
    case 'update':
      return { selectedId: state.selectedId, widgets: update(state.widgets, action['widget'])}
    case 'remove':
      return { selectedId: state.selectedId, widgets: remove(state.widgets, action['widget'])}
    default:
      return state;
  }
}
```

Weak Action Types

```ts
export const loadWidgets = createAction(
  '[Widgets] Load Widgets',
  props<{ widgets: Widget[] }>()
);

export const createWidget = createAction(
  '[Widgets] Create Widget',
  props<{ widget: Widget }>()
);

export const updateWidget = createAction(
  '[Widgets] Update Widget',
  props<{ widget: Widget }>()
);

export const deleteWidget = createAction(
  '[Widgets] Delete Widget',
  props<{ widget: Widget }>()
);
```

Action Creators

```
createWidget(widget: Widget) {
  this.dispatch(WidgetsActions.createWidget({ widget }));
}

updateWidget(widget: Widget) {
  this.dispatch(WidgetsActions.updateWidget({ widget }));
}

deleteWidget(widget: Widget) {
  this.dispatch(WidgetsActions.deleteWidget({ widget }));
}

dispatch(action: Action) {
  this.store.dispatch(action);
}
```

Dispatch Actions

Now we can use **createReducer**

createReducer

- Must be used with an **ActionCreator** and will *not work* with class based action creators
- Depending on your compiler, you will need to **export a reducer function** so AOT will work
- The **createReducer** method takes **initial state** and N number of **on** method calls to handle incoming actions

```
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(WidgetsActions.selectWidget, (state, { selectedId }) => (
    { selectedId, widgets: state.widgets})),
  on(WidgetsActions.loadWidgets, (state, { widgets }) => (
    { selectedId: state.selectedId, widgets})),
  on(WidgetsActions.createWidget, (state, { widget }) =>
    ({ selectedId: state.selectedId, widgets: create(state.widgets, widget)})),
  on(WidgetsActions.updateWidget, (state, { widget }) =>
    ({ selectedId: state.selectedId, widgets: update(state.widgets, widget)})),
  on(WidgetsActions.deleteWidget, (state, { widget }) =>
    ({ selectedId: state.selectedId, widgets: remove(state.widgets, widget)})),
);

export function widgetsReducer(state: WidgetsState = initialWidgetsState, action: Action) {
  return _widgetsReducer(state, action);
}
```

Using createReducer

We can observe dispatched actions using **ActionsSubject**

```
mutations$ = this.actions$.pipe(
  filter((action: Action) =>
    action.type === ItemsActions.createItem({} as any).type ||
    action.type === ItemsActions.updateItem({} as any).type ||
    action.type === ItemsActions.deleteItem({} as any).type
  )
);

constructor(
  private store: Store<fromItems.ItemsPartialState>,
  private actions$: ActionsSubject
) {}
```

Using ActionsSubject

Managing Collections with **Entity**

Entity

- Working with collections can become **expensive** especially when you have to iterate each time you need to make a selection
- **Entity** reduces the amount of boilerplate needed to perform efficient CRUD operations on a collection
- **EntityState** is a predefined generic interface that we extend for our reducer state
- **EntityAdapter** provides a series of methods that we can use on our collection
- **Entity** also provides a numbers of **selectors** for querying our collection

```ts
export interface EntityState<T> {
    ids: string[] | number[];
    entities: Dictionary<T>;
}
```

Entity State

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedId?: string | number; // which Widgets record has been selected
  loaded: boolean; // has the Widgets list been loaded
  error?: string | null; // last known error (if any)
}

export const widgetsAdapter: EntityAdapter<Widget> = createEntityAdapter();

export const initialWidgetsState: WidgetsState = widgetsAdapter.getInitialState({
  // set initial required properties
  loaded: false
});
```

Entity Implementation

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedId?: string | number; // which Widgets record has been selected
  loaded: boolean; // has the Widgets list been loaded
  error?: string | null; // last known error (if any)
}

export const widgetsAdapter: EntityAdapter<Widget> = createEntityAdapter();

export const initialWidgetsState: WidgetsState = widgetsAdapter.getInitialState({
  // set initial required properties
  loaded: false
});
```

Updated Widgets State

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedId?: string | number; // which Widgets record has been selected
  loaded: boolean; // has the Widgets list been loaded
  error?: string | null; // last known error (if any)
}
```

```ts
export const widgetsAdapter: EntityAdapter<Widget> = createEntityAdapter();

export const initialWidgetsState: WidgetsState = widgetsAdapter.getInitialState({
  // set initial required properties
  loaded: false
});
```

Entity Adapter

```
export interface WidgetsState extends EntityState<Widget> {
  selectedId?: string | number; // which Widgets record has been selected
  loaded: boolean; // has the Widgets list been loaded
  error?: string | null; // last known error (if any)
}

export const widgetsAdapter: EntityAdapter<Widget> = createEntityAdapter();

export const initialWidgetsState: WidgetsState = widgetsAdapter.getInitialState({
  // set initial required properties
  loaded: false
});
```

Updated Initial State

```ts
export interface WidgetsState extends EntityState<Widget> {
  selectedId?: string | number; // which Widgets record has been selected
  loaded: boolean; // has the Widgets list been loaded
  error?: string | null; // last known error (if any)
}

export const widgetsAdapter: EntityAdapter<Widget> = createEntityAdapter();

export const initialWidgetsState: WidgetsState = widgetsAdapter.getInitialState({
  // set initial required properties
  loaded: false
});
```

Entity Implementation

```
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(WidgetsActions.selectWidget, (state, { selectedId }) => (
    { selectedId, widgets: state.widgets})),
  on(WidgetsActions.loadWidgets, (state, { widgets }) => (
    { selectedId: state.selectedId, widgets})),
  on(WidgetsActions.createWidget, (state, { widget }) =>
    ({ selectedId: state.selectedId, widgets: create(state.widgets, widget)})),
  on(WidgetsActions.updateWidget, (state, { widget }) =>
    ({ selectedId: state.selectedId, widgets: update(state.widgets, widget)})),
  on(WidgetsActions.deleteWidget, (state, { widget }) =>
    ({ selectedId: state.selectedId, widgets: remove(state.widgets, widget)})),
);
```

Basic Reducer

```ts
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(WidgetsActions.selectWidget, (state, { selectedId }) =>
    Object.assign({}, state, { selectedId })),
  on(WidgetsActions.loadWidgets, (state, { widgets }) =>
    widgetsAdapter.setAll(widgets, { ...state, loaded: true })
  ),
  on(WidgetsActions.createWidget, (state, { widget }) =>
    widgetsAdapter.addOne(widget, state)
  ),
  on(WidgetsActions.updateWidget, (state, { widget }) =>
    widgetsAdapter.updateOne({ id: widget.id, changes: widget }, state)
  ),
  on(WidgetsActions.deleteWidget, (state, { widget }) =>
    widgetsAdapter.removeOne(widget.id, state)
  ),
);
```

Entity Adapter Methods

```
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(WidgetsActions.selectWidget, (state, { selectedId }) =>
    Object.assign({}, state, { selectedId })),
  on(WidgetsActions.loadWidgets, (state, { widgets }) =>
    widgetsAdapter.setAll(widgets, { ...state, loaded: true })
  ),
  on(WidgetsActions.createWidget, (state, { widget }) =>
    widgetsAdapter.addOne(widget, state)
  ),
  on(WidgetsActions.updateWidget, (state, { widget }) =>
    widgetsAdapter.updateOne({ id: widget.id, changes: widget }, state)
  ),
  on(WidgetsActions.deleteWidget, (state, { widget }) =>
    widgetsAdapter.removeOne(widget.id, state)
  ),
);
```

Entity Adapter Methods

```
allWidgets$ = this.store.pipe(
  select('widgets'),
  map(state => data.entities),
  map(state => Object.keys(state).map(k => state[k]))
);
```

Consuming Entities

Querying State with Selectors

Selectors Pt 1

- Selectors allow us to query our application store to obtain a specific slice
- Selectors will recompute if one of its arguments change otherwise it will return the last computed value thanks to memoization
- Selectors are composable and can be used as input to other selectors

```
allWidgets$ = this.store.pipe(
  select('widgets'),
  map(data => data.entities),
  map(data => Object.keys(data).map(k => data[k]))
);
```

Tedious Entity Consumption

```ts
export const getWidgetsState = createFeatureSelector<
  WidgetsState
>(COURSES_FEATURE_KEY);

const { selectAll, selectEntities } = widgetsAdapter.getSelectors();

export const getAllWidgets = createSelector(
  getWidgetsState,
  (state: WidgetsState) => selectAll(state)
);
```

Selector Setup

```ts
export const getWidgetsState = createFeatureSelector<
  WidgetsState
>(COURSES_FEATURE_KEY);

const { selectAll, selectEntities } = widgetsAdapter.getSelectors();

export const getAllWidgets = createSelector(
  getWidgetsState,
  (state: WidgetsState) => selectAll(state)
);
```

Feature Selector

```
export const getWidgetsState = createFeatureSelector<
  WidgetsState
>(COURSES_FEATURE_KEY);

const { selectAll, selectEntities } = widgetsAdapter.getSelectors();

export const getAllWidgets = createSelector(
  getWidgetsState,
  (state: WidgetsState) => selectAll(state)
);
```

Entity Adapter Selectors

```
export const getWidgetsState = createFeatureSelector<
  WidgetsState
>(COURSES_FEATURE_KEY);

const { selectAll, selectEntities } = widgetsAdapter.getSelectors();

export const getAllWidgets = createSelector(
  getWidgetsState,
  (state: WidgetsState) => selectAll(state)
);
```

Get All Widgets

```ts
import * as fromWidgets from './widgets.reducer';
import * as WidgetsActions from './widgets.actions';
import * as WidgetsSelectors from './widgets.selectors';

@Injectable({
  providedIn: 'root'
})
export class WidgetsFacade {
  allWidgets$ = this.store.pipe(select(WidgetsSelectors.getAllWidgets));

  constructor(
    private store: Store<fromWidgets.WidgetsPartialState>
  ) { }
}
```

Selector Consumption

Asynchronous Operations with **Effects**

Event

Async
Operation

Reducer

Event **Service** **Reducer**

Event

Effect

Reducer

Effects

- Effects is the middleware used to handle asynchronous operations
- It captures an initial event, performs an async operation and then fires off a completion event for the reducer to handle
- Effects allow async services to focus entirely on server communication without having to worry about managing state
- Because effects are observable streams, we have all the power of RxJS at our disposal to perform complex data manipulation if we need to

Trigger Event

Async Operation

Result Event

```ts
export const updateWidget = createAction(
  '[Widgets] Update Widget',
  props<{ widget: Widget }>()
);

export const updateWidgetSuccess = createAction(
  '[Widgets] Update Widget Success',
  props<{ widget: Widget }>()
);

export const updateWidgetFailure = createAction(
  '[Widgets] Update Widget Failure',
  props<{ error: any }>()
);
```

Action Pairs

```ts
// This doesn't work...
loadWidgets() {
  this.widgetsService.all()
    .subscribe((widgets: Widget[]) =>
      this.dispatch(WidgetsActions.loadWidgets({ widgets })));
}
```

```ts
// This does...
loadWidgets() {
  this.widgetsService.all()
    .subscribe((widgets: Widget[]) =>
      this.dispatch(WidgetsActions.loadWidgetsSuccess({ widgets })));
}
```

Updated Action

```
loadWidgets$ = createEffect(() => this.actions$.pipe(
  ofType(WidgetsActions.loadWidgets),
  fetch({
    run: (action) =>
      this.widgetsService
        .all()
        .pipe(
          map((widgets: Widget[]) =>
            WidgetsActions.loadWidgetsSuccess({ widgets })
          )
        ),
    onError: (action, error) => WidgetsActions.loadWidgetsFailure({ error }),
  })
));
```

Effect

```
loadWidgets$ = createEffect(() => this.actions$.pipe(
  ofType(WidgetsActions.loadWidgets),
  fetch({
    run: (action) =>
      this.widgetsService
        .all()
        .pipe(
          map((widgets: Widget[]) =>
            WidgetsActions.loadWidgetsSuccess({ widgets })
          )
        ),
    onError: (action, error) => WidgetsActions.loadWidgetsFailure({ error }),
  })
));
```

Trigger Event

```
loadWidgets$ = createEffect(() => this.actions$.pipe(
  ofType(WidgetsActions.loadWidgets),
  fetch({
    run: (action) =>
      this.widgetsService
        .all()
        .pipe(
          map((widgets: Widget[]) =>
            WidgetsActions.loadWidgetsSuccess({ widgets })
          )
        ),
    onError: (action, error) => WidgetsActions.loadWidgetsFailure({ error }),
  })
));
```

Result Events

```ts
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(WidgetsActions.selectWidget, (state, { selectedId }) =>
    Object.assign({}, state, { selectedId })),
  on(WidgetsActions.loadWidgetsSuccess, (state, { widgets }) =>
    widgetsAdapter.setAll(widgets, { ...state, loaded: true })
  ),
  on(WidgetsActions.createWidgetSuccess, (state, { widget }) =>
    widgetsAdapter.addOne(widget, state)
  ),
  on(WidgetsActions.updateWidgetSuccess, (state, { widget }) =>
    widgetsAdapter.updateOne({ id: widget.id, changes: widget }, state)
  ),
  on(WidgetsActions.deleteWidgetSuccess, (state, { widget }) =>
    widgetsAdapter.removeOne(widget.id, state)
  ),
);
```

Updated Reducer

```
const _widgetsReducer = createReducer(
  initialWidgetsState,
  on(WidgetsActions.selectWidget, (state, { selectedId }) =>
    Object.assign({}, state, { selectedId })),
  on(WidgetsActions.loadWidgetsSuccess, (state, { widgets }) =>
    widgetsAdapter.setAll(widgets, { ...state, loaded: true })
  ),
  on(WidgetsActions.createWidgetSuccess, (state, { widget }) =>
    widgetsAdapter.addOne(widget, state)
  ),
  on(WidgetsActions.updateWidgetSuccess, (state, { widget }) =>
    widgetsAdapter.updateOne({ id: widget.id, changes: widget }, state)
  ),
  on(WidgetsActions.deleteWidgetSuccess, (state, { widget }) =>
    widgetsAdapter.removeOne(widget.id, state)
  ),
);
```

Updated Reducer

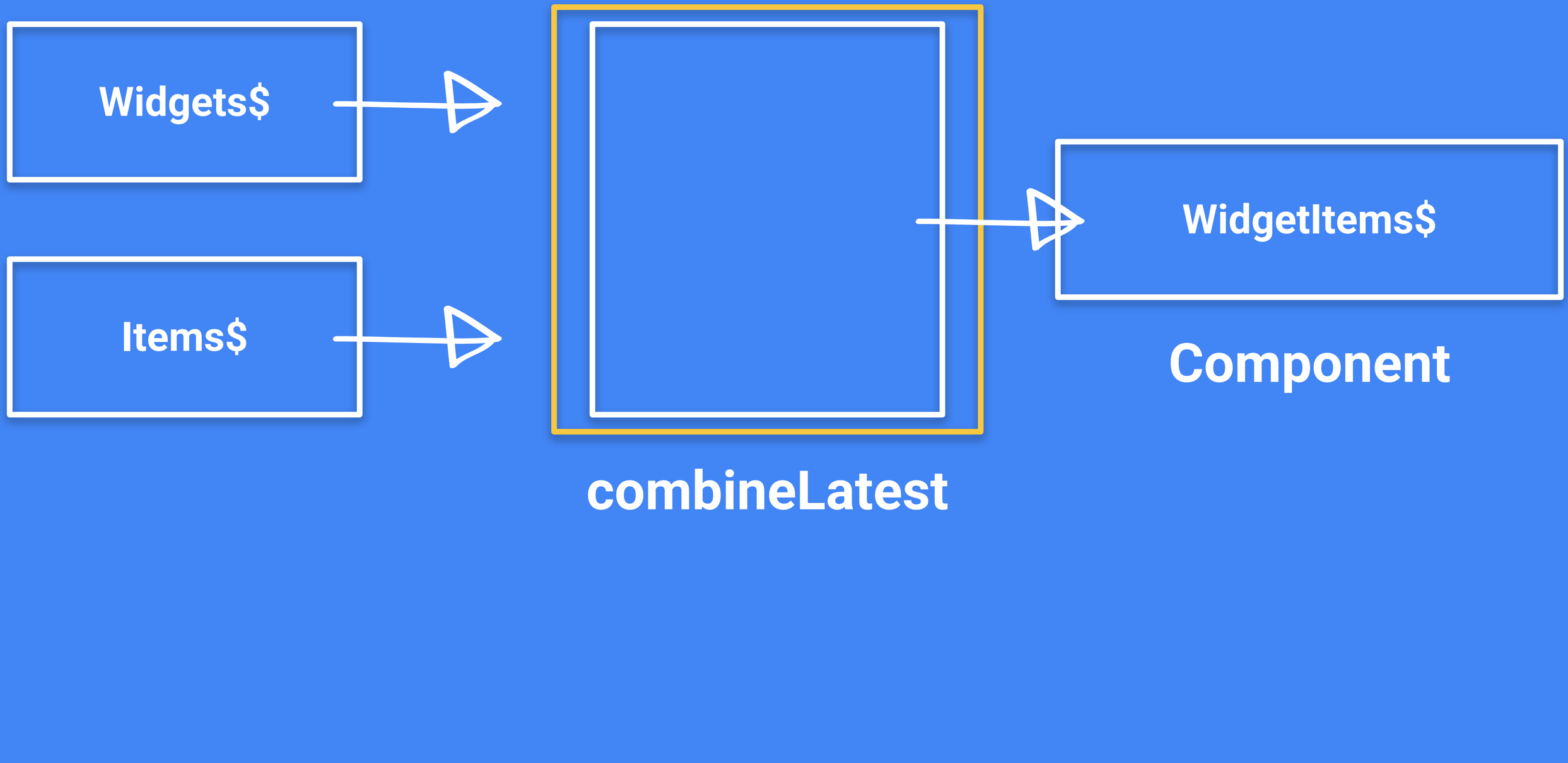
```ts
export class WidgetsService {
  model = 'widgets';

  constructor(private http: HttpClient) { }

  all() { return this.http.get(this.getUrl()); }
  find(id: string) { return this.http.get(this.getUrlWithId(id));  }
  create(widget: Widget) { return this.http.post(this.getUrl(), widget); }
  update(widget: Widget) { return this.http.put(this.getUrlWithId(widget.id), widget); }
  delete(id: string) { return this.http.delete(this.getUrlWithId(id)); }

  private getUrl() { return `${environment.apiEndpoint}${this.model}`; }
  private getUrlWithId(id) { return `${this.getUrl()}/${id}`; }
}
```

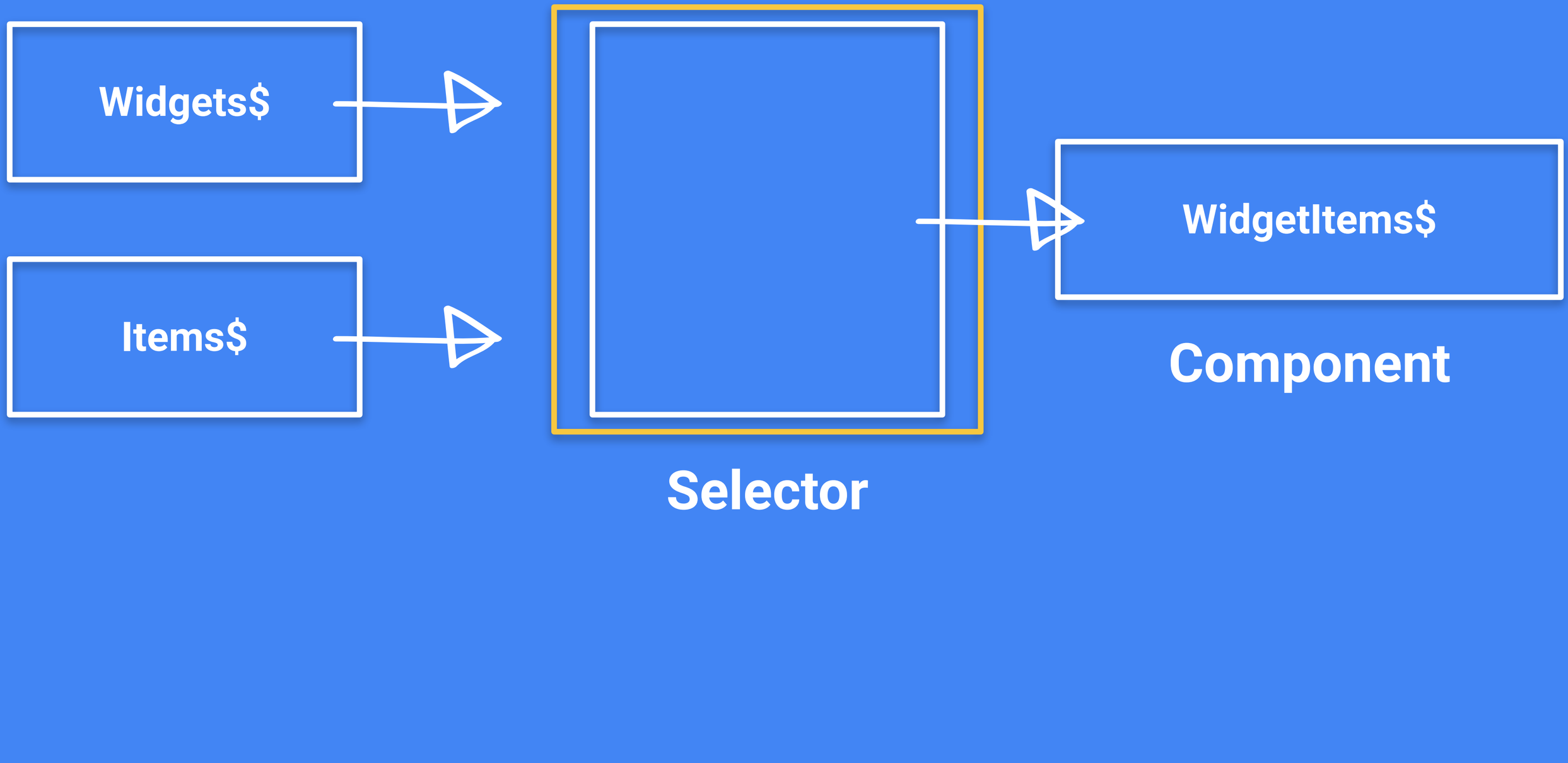
Concise Services FTW!

```
// Ain't nobody got time for this...
loadWidgets() {
  this.widgetsService.all()
    .subscribe((widgets: Widget[]) =>
      this.dispatch(WidgetsActions.loadWidgetsSuccess({ widgets })));
}

// when you can have this...
loadWidgets() {
  this.dispatch(WidgetsActions.loadWidgets())
}
```

Updated Facade

Computed Data with Selectors


Widgets$
Items$
combineLatest
WidgetItems$
Component


Widgets$
Items$
WidgetItems$
Component
Selector

Selectors Pt 2

- Selectors can compute derived data
- Selectors are not recomputed unless one of its arguments change
- Selectors are composable and can be used as input to other selectors

```ts
export const getWidgetsEntities = createSelector(
  getWidgetsState,
  (state: WidgetsState) => selectEntities(state)
);

export const getSelectedWidgetId = createSelector(
  getWidgetsState,
  (state: WidgetsState) => state.selectedId
);

export const getSelectedWidget = createSelector(
  getWidgetsEntities,
  getSelectedWidgetId,
  (entities, selectedId) => selectedId && entities[selectedId]
);
```

Combined Selector

```ts
import * as fromWidgets from './widgets.reducer';
import * as WidgetsActions from './widgets.actions';
import * as WidgetsSelectors from './widgets.selectors';

@Injectable({
  providedIn: 'root'
})
export class WidgetsFacade {
  allWidgets$ = this.store.pipe(select(WidgetsSelectors.getAllWidgets));
  selectedWidget$ = this.store.pipe(select(WidgetsSelectors.getSelectedWidget));

  constructor(
    private store: Store<fromWidgets.WidgetsPartialState>
  ) { }
}
```

Selector Consumption

```ts
const emptyWidget: Widget = { id: null, title: '', description: '' };

export const getSelectedWidget = createSelector(
  getWidgetsEntities,
  getSelectedWidgetId,
  (entities, selectedId) => {
    return selectedId ? entities[selectedId] : emptyWidget;
  }
);
```

Selector with Computation

```ts
export const getWidgetItems = createSelector(
  WidgetsSelectors.getAllWidgets,
  ItemsSelectors.getAllItems,
  (widgets: Widget[], items: Item[]) => {
    return widgets.map((widget) => ({
      ...widget,
      items: items.filter((item) => item.widget_id === widget.id),
    }));
  }
);
```

Computed Models

INTERMISSION