**TypeScript**

* **What is TypeScript?**
  + Superset of JavaScript
  + Statically-typed language
  + Compiles to plain JavaScript
  + Adds optional static type system to JavaScript
* **Why use TypeScript?**
  + **Improved Code Quality:** Catches errors early (compile-time)
  + **Better Tooling:** Autocomplete, refactoring, navigation
  + **Readability & Maintainability:** Explicit types make code easier to understand
  + **Scalability:** Ideal for large-scale applications
  + **Future-Proofing:** Aligns with modern JavaScript features
* **Core Concepts:**
  + **1. Types**
    - **Basic Types:**
      * number (e.g., 10, 3.14)
      * string (e.g., "hello")
      * boolean (e.g., true, false)
      * any (disables type-checking, use with caution)
      * void (for functions that return nothing)
      * null and undefined
    - **Advanced Types:**
      * Array (e.g., number[], string[])
      * Tuple (fixed-size array with specific types, e.g., [string, number])
      * Enum (set of named constants, e.g., enum Color { Red, Green, Blue })
      * Object
      * Literal Types (e.g., "red", 42)
  + **2. Type Annotations & Inference**
    - **Annotation:** Explicitly specifying the type of a variable, function parameter, or return value (e.g., let age: number = 25;)
    - **Inference:** TypeScript automatically deduces the type if you don't explicitly annotate it (e.g., let name = "Alice"; -> name is inferred as string).
  + **3. Interfaces**
    - **Purpose:** Define the "shape" of an object.
    - **Usage:** Enforce a contract for objects, classes, and function parameters.
    - **Features:** Optional properties (?), readonly properties, extending other interfaces.
    - **Example:**

TypeScript

interface User {

id: number;

name: string;

email?: string; // Optional

}

* + **4. Classes**
    - **Purpose:** Define blueprints for creating objects.
    - **Features:**
      * Properties and methods
      * Access modifiers: public, private, protected
      * Constructors
      * Inheritance (extends)
      * Implementing Interfaces (implements)
  + **5. Functions**
    - **Purpose:** Define reusable blocks of code with type safety.
    - **Features:**
      * Type annotations for parameters and return values.
      * Optional parameters (?)
      * Default parameters
      * Rest parameters
      * Function overloads (multiple function signatures for the same name)
  + **6. Generics**
    - **Purpose:** Create reusable components that work with a variety of types.
    - **Usage:** Used for functions, classes, and interfaces.
    - **Example:**

TypeScript

function identity<T>(arg: T): T {

return arg;

}

* + **7. Modules**
    - **Purpose:** Organize code into separate, reusable files.
    - **Concepts:**
      * export: Make a variable, function, or class available to other files.
      * import: Use exported components from other files.
    - **Types:** ES Modules (import/export), CommonJS (require/module.exports).
* **Build Process**
  + **tsc (TypeScript Compiler):**
    - Command-line tool to compile .ts files to .js files.
  + **tsconfig.json:**
    - Configuration file for the tsc compiler.
    - Defines compiler options (target, module, strict, outDir, etc.).
* **Advanced Topics (for further study)**
  + **Union Types:** string | number
  + **Intersection Types:** TypeA & TypeB
  + **Type Aliases:** type MyString = string;
  + **keyof Operator:** Get union of property names from a type.
  + **Type Guards:** Techniques to narrow down a type (e.g., typeof, instanceof).
  + **Decorators**
  + **Type Declaration Files (.d.ts)**

**Angular 17**

* **Key Philosophy:**
  + Developer Experience (DX) and Performance
  + Moving away from boilerplate and complexity
  + Standalone-first, module-optional
  + Modern toolchain (Vite & esbuild)
* **Core Building Blocks:**
  + **1. Standalone Components, Directives, & Pipes**
    - **Purpose:** Reduce reliance on NgModules for a simpler, more modular structure.
    - **How it works:**
      * standalone: true in the @Component decorator.
      * Dependencies are imported directly via the imports array.
      * This simplifies the project structure, especially for small to medium-sized applications.
    - **Impact:**
      * No more declaring components in a module.
      * Easier lazy loading and code sharing.
      * bootstrapApplication() is used instead of bootstrapping a root module.
  + **2. New Built-in Control Flow**
    - **Purpose:** A new, more ergonomic and performant syntax for template logic.
    - **Syntax:**
      * @if and @else for conditional rendering.
      * @for for loops, which *requires* a track function for optimized re-rendering.
      * @switch and @case for complex conditional logic.
    - **Impact:**
      * Replaces \*ngIf, \*ngFor, and \*ngSwitch directives.
      * Improves readability and matches JavaScript syntax.
      * Better performance and reduced bundle size (up to 30kb).
  + **Routing:**
    - The router works seamlessly with standalone components.
    - Lazy loading can now be done without modules.
  + **Dependency Injection (DI):**
    - Still a core concept for providing services.
    - Standalone components provide their services directly.
    - provideRouter and other provide-prefixed functions replace RouterModule.forRoot and similar module-based setups.
  + **Server-Side Rendering (SSR) & Static Site Generation (SSG):**
    - Stable and easier to set up.
    - Hydration is enabled by default to re-use server-rendered HTML.
    - ng new --ssr command for new projects.
    - Improves Core Web Vitals (CWV) and SEO.
  + **Build Process:**
    - **Vite & esbuild:** The new default build tools for faster development and production builds.
    - Significant improvements in build time (up to 67% faster).
    - ng serve and ng build are now powered by this modern toolchain.
* **Project Structure (ng new)**
  + New projects are standalone by default.
  + src/app/app.config.ts: Central place for application-level providers.
  + src/main.ts: bootstrapApplication(AppComponent, appConfig).
  + This structure is leaner and more straightforward.

**Central Topic: Angular Forms**

* **Purpose:** Handle user input and data interaction within an Angular application.
* **Core Goal:** Manage the state of form controls (validity, value, touched/dirty state).
* **Branch 1: Template-Driven Forms**
  + **Philosophy:**
    - Rely on directives and attributes in the component's HTML template.
    - Minimal code in the component class.
    - Best for simple forms with straightforward validation.
  + **Key Directives:**
    - FormsModule: The module that needs to be imported to enable template-driven forms.
    - ngForm: Automatically created on <form> elements with a name attribute. Manages the form as a whole.
    - ngModel: The core directive for two-way data binding ([(ngModel)]). Binds a control to a component property.
    - ngModelGroup: Groups a set of controls together (e.g., for an address block).
  + **Validation:**
    - **Built-in Validators (HTML5 attributes):**
      * required
      * minlength, maxlength
      * pattern
    - **CSS Classes:** Angular adds classes to elements based on their state:
      * .ng-valid, .ng-invalid
      * .ng-pristine, .ng-dirty
      * .ng-untouched, .ng-touched
    - **Accessing Form State:** Use a template reference variable (#myForm="ngForm") to access the NgForm object and its properties (e.g., myForm.valid).
  + **How it Works (internally):**
    - Angular automatically creates an FormControl instance for each ngModel directive.
    - It creates an NgForm instance for the entire <form>.
    - Changes in the template update the model, and changes in the model update the template (two-way binding).
* **Branch 2: Reactive Forms**
  + **Philosophy:**
    - Manage the form's state directly in the component class.
    - Code-driven, programmatic approach.
    - Best for complex forms, dynamic forms, and custom validation.
  + **Key Classes:**
    - ReactiveFormsModule: The module that needs to be imported.
    - FormGroup: A collection of FormControls. Represents the entire form.
    - FormControl: Represents a single input field.
    - FormArray: A collection of FormGroups or FormControls (e.g., for a list of dynamic items).
  + **Creating a Form:**
    - **Step 1:** Import FormGroup and FormControl from @angular/forms.
    - **Step 2:** Instantiate a FormGroup in the component class.
      * const myForm = new FormGroup({ ... });
    - **Step 3:** Define a FormControl for each input field within the FormGroup.
      * name: new FormControl('initial value', [Validators.required])
    - **Step 4:** In the template, bind the HTML <form> to the FormGroup using [formGroup]="myForm".
    - **Step 5:** Bind individual input fields using formControlName="name".
  + **Validation:**
    - **Validators:** Static functions imported from @angular/forms.
    - **Built-in Validators:** Validators.required, Validators.minLength, Validators.maxLength, Validators.pattern, Validators.email.
    - **Custom Validators:** Write your own functions to perform specific validation logic.
    - **Asynchronous Validators:** For validation that requires a server request.
  + **Dynamic Forms:**
    - Easily add or remove controls or entire form groups programmatically.
    - FormArray is the key for dynamic lists.
    - myForm.addControl('newControl', new FormControl());
* **Branch 3: Common Concepts & Best Practices**
  + **Validation Messages:**
    - Use \*ngIf to show/hide validation error messages.
    - Check for form state properties like control.invalid and control.touched.
  + **Form Submission:**
    - Use the (ngSubmit) event in the template.
    - Prevent default browser behavior.
    - In Reactive Forms, use myForm.value to get the form data.
  + **State Management:**
    - **value:** The current value of the form.
    - **status:** VALID, INVALID, PENDING, DISABLED.
    - **dirty / pristine:** Has the user changed the value?
    - **touched / untouched:** Has the user visited and left the field?
  + **Form Builder Service:**
    - A helper service that simplifies the creation of FormGroups, FormControls, and FormArrays.
    - Shorter, more readable syntax: this.fb.group({ ... }).
    - Recommended for complex Reactive Forms.

**Central Topic: Bootstrap Grid**

* **Purpose:** Create responsive, mobile-first layouts using a series of containers, rows, and columns.
* **Core Philosophy:**
  + "Columns-in-Rows-in-Containers"
  + Mobile-first design: Starts with the smallest screen size and scales up.
  + 12-column system.
* **Branch 1: Key Components**
  + **1. Containers (.container and .container-fluid)**
    - **.container:**
      * A fixed-width container.
      * Provides padding and centers the content.
      * Width changes at each responsive breakpoint (sm, md, lg, xl, xxl).
      * Best for centered, non-full-width layouts.
    - **.container-fluid:**
      * A full-width container (width: 100%).
      * Stretches to fill the entire viewport width.
      * Best for layouts that span the full width of the screen.
  + **2. Rows (.row)**
    - **Purpose:** A wrapper for columns.
    - **Key Feature:** Uses CSS Flexbox (display: flex).
    - **Function:**
      * Creates a horizontal group of columns.
      * Provides negative margins to offset the padding of the columns, ensuring content alignment.
  + **3. Columns (.col-\*)**
    - **Purpose:** The building blocks that hold your content.
    - **Key Feature:** Fit within a .row.
    - **Rules:**
      * A row can contain up to 12 columns.
      * If you add more than 12, they will wrap to a new line.
      * You can group columns to achieve different widths (e.g., three .col-4 columns).
* **Branch 2: Responsive Breakpoints**
  + **Purpose:** Define screen size ranges to control how the layout behaves.
  + **How it works:** Grid classes are suffixed with an abbreviation for a specific breakpoint.
  + **Breakpoints (from smallest to largest):**
    - **xs (Extra Small):**
      * <576px
      * No suffix needed for mobile-first approach (e.g., .col-4).
    - **sm (Small):**
      * ≥576px
      * Suffix: sm (e.g., .col-sm-6).
    - **md (Medium):**
      * ≥768px
      * Suffix: md (e.g., .col-md-4).
    - **lg (Large):**
      * ≥992px
      * Suffix: lg (e.g., .col-lg-3).
    - **xl (Extra Large):**
      * ≥1200px
      * Suffix: xl (e.g., .col-xl-2).
    - **xxl (Extra Extra Large):**
      * ≥1400px
      * Suffix: xxl (e.g., .col-xxl-1).
* **Branch 3: Column Classes & Variations**
  + **1. Column Widths (.col-{breakpoint}-{width})**
    - **Syntax:** col-md-6
    - **Meaning:** On medium screens and up, the column will take up 6 of the 12 available columns.
    - **Un-suffixed:** A class like .col-6 will apply to all screen sizes, from xs up.
  + **2. Automatic Layout (.col, .col-{breakpoint})**
    - **Purpose:** Create columns of equal width.
    - **How it works:**
      * .col: Each column will automatically get an equal share of the row's space.
      * .col-sm: Equal width on small screens and up.
  + **3. Offsetting Columns (.offset-{breakpoint}-{width})**
    - **Purpose:** Move a column to the right by a specified number of columns.
    - **Syntax:** .offset-md-3
    - **Meaning:** Pushes the column to the right by 3 column units on medium screens and up.
  + **4. Reordering (.order-{number}, .order-{breakpoint}-{number})**
    - **Purpose:** Change the visual order of columns, independent of their HTML source order.
    - **Syntax:** .order-1, .order-lg-3
    - **How it works:** Uses Flexbox order property.
  + **5. Nesting Grids**
    - **Purpose:** Create more complex layouts.
    - **How it works:** Place a new .row and a set of .col-\* classes inside an existing .col-\* column. The nested row also has 12 columns.
* **Branch 4: Alignment & Spacing**
  + **1. Vertical Alignment (.align-items-\*)**
    - **Purpose:** Control vertical alignment of columns within a row.
    - **Classes:** .align-items-start, .align-items-center, .align-items-end.
  + **2. Horizontal Alignment (.justify-content-\*)**
    - **Purpose:** Control horizontal alignment of columns within a row.
    - **Classes:** .justify-content-start, .justify-content-center, .justify-content-end, .justify-content-between, .justify-content-around.
  + **3. Gutters (Padding & Margins)**
    - **Purpose:** Control the space between columns.
    - **Classes:** Use gutter classes like g-3 or gx-2 and gy-4 on the .row element to set spacing.
    - **How it works:** Uses CSS custom properties to control the horizontal and vertical gutters.

**Central Topic: NgRx State Management**

* **What is NgRx?**
  + A library for managing application state.
  + Inspired by Redux, it follows a unidirectional data flow.
  + Provides a predictable and scalable way to manage state, especially in large applications.
  + Built on top of RxJS for handling streams of data.
* **Branch 1: Core Concepts & Unidirectional Data Flow**
  + **1. State:**
    - A single, immutable JavaScript object representing the entire application's state.
    - The "Single Source of Truth."
  + **2. Store (@ngrx/store):**
    - The central hub that holds the State.
    - Provides methods to dispatch Actions and select slices of the State.
  + **3. Data Flow (Mental Model):**
    - **1. Component Dispatches an Action:** A user interaction triggers an Action.
    - **2. Effects Listen for Actions:** Effects intercept the Action and perform a side effect.
    - **3. Effects Dispatches a New Action:** After the side effect (e.g., API call) is complete, a new success or failure Action is dispatched.
    - **4. Reducers Listen for Actions:** The Reducer takes the Action and the current State.
    - **5. Reducers Return a New State:** The Reducer calculates and returns a **new, immutable** State object.
    - **6. Selectors Read from the Store:** Components subscribe to Selectors to get the updated State and render the UI.
* **Branch 2: Building Blocks (@ngrx/store)**
  + **1. Actions:**
    - **Purpose:** Describe unique events that happen in the application.
    - **Structure:** A class or object with a type property (e.g., [Products Page] Load Products).
    - **Types of Actions:**
      * **Input Actions:** Triggered by user interaction (e.g., [Cart] Add Item).
      * **Output Actions:** Triggered by an Effect (e.g., [API] Load Products Success).
      * createAction() from @ngrx/store is the modern way to define them.
  + **2. Reducers:**
    - **Purpose:** A pure function that takes the current State and an Action and returns a **new** State.
    - **Rules:** Must be **pure** (no side effects, no mutation).
    - **Function:** (state, action) => newState.
    - createReducer() from @ngrx/store is the recommended method.
  + **3. Selectors:**
    - **Purpose:** Functions for retrieving specific, small slices of the State from the Store.
    - **Benefits:**
      * **Performance:** They are memoized, so they only re-run when their inputs change.
      * **Encapsulation:** Components don't need to know the entire state structure.
    - **How to create:** Use createSelector() from @ngrx/store.
* **Branch 3: Effects (@ngrx/effects)**
  + **Purpose:** Handle side effects that are not part of the pure state change, such as:
    - **API Calls:** Fetching data from a server.
    - **Database Operations:** Saving data to a local database.
    - **Authentication:** Logging in or out.
    - **Router Navigation:** Navigating to a new route.
  + **Mechanism:**
    - A class decorated with @Injectable() that listens to a stream of Actions.
    - It uses RxJS operators (like ofType, exhaustMap, mergeMap, catchError) to perform asynchronous operations.
    - **The key rule:** After a side effect is complete, it **must** dispatch a new Action.
  + **Example Flow:**
    - **Component:** dispatch(LoadProductsAction)
    - **Effect:**
      * ofType(LoadProductsAction) listens for the action.
      * Calls a service method (productService.getProducts()).
      * On success, dispatches LoadProductsSuccessAction.
      * On error, dispatches LoadProductsFailureAction.
    - **Reducer:** Listens for LoadProductsSuccessAction and updates the State with the new product data.
* **Branch 4: Best Practices & Utilities**
  + **1. NgRx Schematics:**
    - Use ng add @ngrx/store to set up the environment.
    - Use ng generate commands (e.g., ng g store, ng g action, ng g reducer, ng g effect) to quickly scaffold code.
  + **2. Store Devtools (@ngrx/store-devtools):**
    - A powerful Chrome/Firefox extension for debugging NgRx.
    - Allows you to time-travel, inspect state changes, and see a history of all dispatched Actions.
  + **3. State Normalization:**
    - A pattern for storing data in a flat structure, similar to a database.
    - Prevents data duplication and simplifies updates.
    - Use @ngrx/entity to simplify working with normalized state.
  + **4. Feature State:**
    - Organize your state into modules (e.g., products, auth, cart).
    - This keeps the state tree manageable and scalable.
    - Use StoreModule.forFeature() to lazy-load feature modules.
  + **5. Strict Immutability:**
    - Always return a new object from Reducers.
    - **Never** mutate the existing State object directly. NgRx will not detect the change.
    - Use the spread operator (...) to create a new object with updated properties.

**1. Authentication (Who are you?)**

* **User Credentials:**
  + Login/Registration forms
  + Username/password
* **Authentication Service:**
  + Handles login, logout, and token management.
  + Communicates with a backend API.
  + Stores and retrieves authentication state (e.g., using localStorage or sessionStorage).
* **Token-Based Authentication:**
  + **JSON Web Tokens (JWT):** A popular method for securely transmitting information between parties as a JSON object.
  + HTTP Interceptors: Automatically attach the JWT to all outgoing HTTP requests to protected resources.

**2. Authorization (What are you allowed to do?)**

* **Role-Based Access Control (RBAC):**
  + Assigning roles (e.g., admin, moderator, user) to authenticated users.
  + Determining access based on the user's assigned roles.
* **Permission-Based Access Control:**
  + Granting specific permissions to perform actions (e.g., canEditProduct, canDeleteUser).
* **Route Guards:**
  + **CanActivate:** Prevents unauthorized navigation to a route.
  + **CanActivateChild:** Prevents unauthorized navigation to child routes.
  + **CanLoad:** Prevents the lazy-loading of modules for unauthorized users.
* **Securing Components & Services:**
  + Injecting an Authorization Service into components.
  + Using structural directives like \*ngIf to hide or show elements based on user roles or permissions.
  + Checking user permissions before performing actions in services.

**3. Key Concepts & Best Practices**

* **Services:** Centralize authentication and authorization logic in dedicated services.
* **Guards:** Use route guards to protect your application's routes.
* **Interceptors:** Use HTTP interceptors to handle the automatic attachment of authentication tokens.
* **Secure Storage:** Carefully choose where to store sensitive information like tokens (e.g., HttpOnly cookies for increased security).
* **User Experience:** Redirect unauthorized users to a login page or an "unauthorized" page.
* **State Management:** Use a state management solution (like a BehaviorSubject in a service) to keep the authentication state synchronized across the application.