newfire
global partners

Angular vs Svelte

Which Is Better for Web Development

Angular vs Svelte

Introduction

Angular and Svelte are both cutting-edge JavaScript frameworks used to create web applications. Developed and maintained by Google, Angular is an established framework, while Svelte has emerged as a popular choice more recently. Despite each having its own set of advantages and drawbacks, they differ significantly in several key aspects.

Angular offers a comprehensive framework, equipping developers with a plethora of tools and features for constructing intricate applications. It employs a component-based architecture and facilitates two-way data binding, ensuring that modifications made in the view are automatically mirrored in the model and vice versa. Additionally, Angular encompasses features such as dependency injection, routing, and animations, making it a potent tool for building complex applications."

Conversely, Svelte is a streamlined framework that emphasizes performance and simplicity. It adopts a unique approach to rendering compared to other frameworks like React and Angular. Rather than rendering the entire app on the client-side, Svelte compiles the app into highly efficient JavaScript code tailored for performance. Consequently, Svelte apps operate faster and consume less memory than those built with other frameworks.

Overview: Advantages & Disadvantages of Both Angular & Svelte

Angular

Advantages:

- Offers a comprehensive framework with numerous features and tools.
- Simplifies development with two-way data binding.
- Well-established with a vast community.
- Excellent compatibility with TypeScript.

Disadvantages:

- Could be challenging to learn, particularly for those new to web development.
- Large codebase might result in slower app performance.
- Might be too complex for basic applications.

Svelte

Advantages:

- A lightweight framework designed for top performance.
- User-friendly and simple to learn, especially for those acquainted with web development.
- Smaller codebase allows for quicker app performance.
- Compatible with TypeScript.

Disadvantages:

- Being relatively new, it has less community support and fewer available resources.
- Missing some features found in other frameworks, such as routing and animations.
- Not as well-suited for complex applications as Angular or React.

Feature	Angular	Svelte
Size	Large (~500KB)	Small (~10KB)
Performance	Slower due to large codebase and two-way binding	Faster due to lightweight framework and compiled code
Learning curve	Challenging, especially for new developers	Fairly easy for web developers
Community support	Large, well-established community	Smaller community, but growing
Templating	HTML-based	Custom language
State management	Reactive programming with RxJS	Built-in state management
Dependency injection	Built-in	Not built-in, but can be achieved with libraries
Routing	Built-in	Not built-in, but can be achieved with libraries
Animations	Built-in	Not built-in, but can be achieved with libraries
Testing	Built-in	Not built-in, but can be achieved with libraries
Type checking	Optional with TypeScript	Optional with TypeScript

As you can see, Angular and Svelte have several key differences. Angular is a comprehensive, feature-rich framework that may be overwhelming for new developers, while Svelte is a lightweight framework that prioritizes performance and simplicity but might lack some features found in other frameworks.

One major difference between the two frameworks is their rendering approach. Angular uses a template-based method with HTML templates defining the application's view, while Svelte employs a custom language compiled into highly efficient JavaScript code for improved performance.

Another crucial aspect to consider is the framework size. Angular, being a large framework, could be excessive for small or straightforward applications. In contrast, Svelte is a lightweight option, ideal for small or medium-sized projects.

In terms of state management, Angular adopts reactive programming with RxJS, while Svelte features built-in state management. Both frameworks support type checking with TypeScript, which helps identify errors early during development.

The size of a framework can influence the time it takes for a user to load a web application. Generally, a smaller framework leads to faster load times and an improved user experience. Let's examine how Angular and Svelte compare in terms of size.

Framework	Size (minified and gzipped)
Angular	~90KB
Svelte	~10KB

As we can see, Svelte is considerably smaller than Angular, potentially leading to quicker load times and enhanced performance.

The performance of a web application also affects the user experience, with a faster application generally resulting in a more responsive and seamless experience for the user. Various factors can impact performance, including the framework size, rendering approach, and code efficiency.

To compare Angular and Svelte's performance, we'll examine benchmarks for different scenarios, including startup time, update time, and memory usage.
Startup time measures the duration it takes for an application to start up and load the initial view. We can use the js-framework-benchmark tool to measure startup time. Here are the results for Angular and Svelte:

Framework	Score (lower is better)
Angular	4.4 seconds
Svelte	0.65 seconds

As shown, Svelte significantly outperforms Angular in terms of startup time.

Update time measures the time required for the view to update when data changes. We can use the js-framework-benchmark tool to measure update time. Here are the results for Angular and Svelte:

Framework	Score (lower is better)
Angular	13.8ms
Svelte	4.4ms

As we can see, Svelte significantly outperforms Angular in terms of update time.

Memory usage measures the amount of memory an application consumes. We can use Chrome DevTools to measure memory usage. Here are the results for Angular and Svelte:

Framework	Memory usage (MB)
Angular	7.9 MB
Svelte	2.5 MB

As demonstrated, Svelte consumes considerably less memory than Angular, potentially resulting in better performance.

Ease of Learning

The ease of learning a framework can affect how quickly and effortlessly a development team can become skilled at using it. A more challenging learning process may require additional time and effort for training and familiarizing new team members. Let's examine how Angular and Svelte compare in terms of the ease of learning.

Framework	Ease of Learning
Angular	Challenging
Svelte	Relatively easy

As we can see, Svelte has a gentler learning curve than Angular, making it more accessible for developers new to web development.

Community Support

The extent of community support can influence a framework's success, as it offers access to resources, assistance, and a network of fellow developers using the same technology. A larger, more active community may also lead to more frequent updates and bug fixes. Let's compare Angular and Svelte in terms of community support.

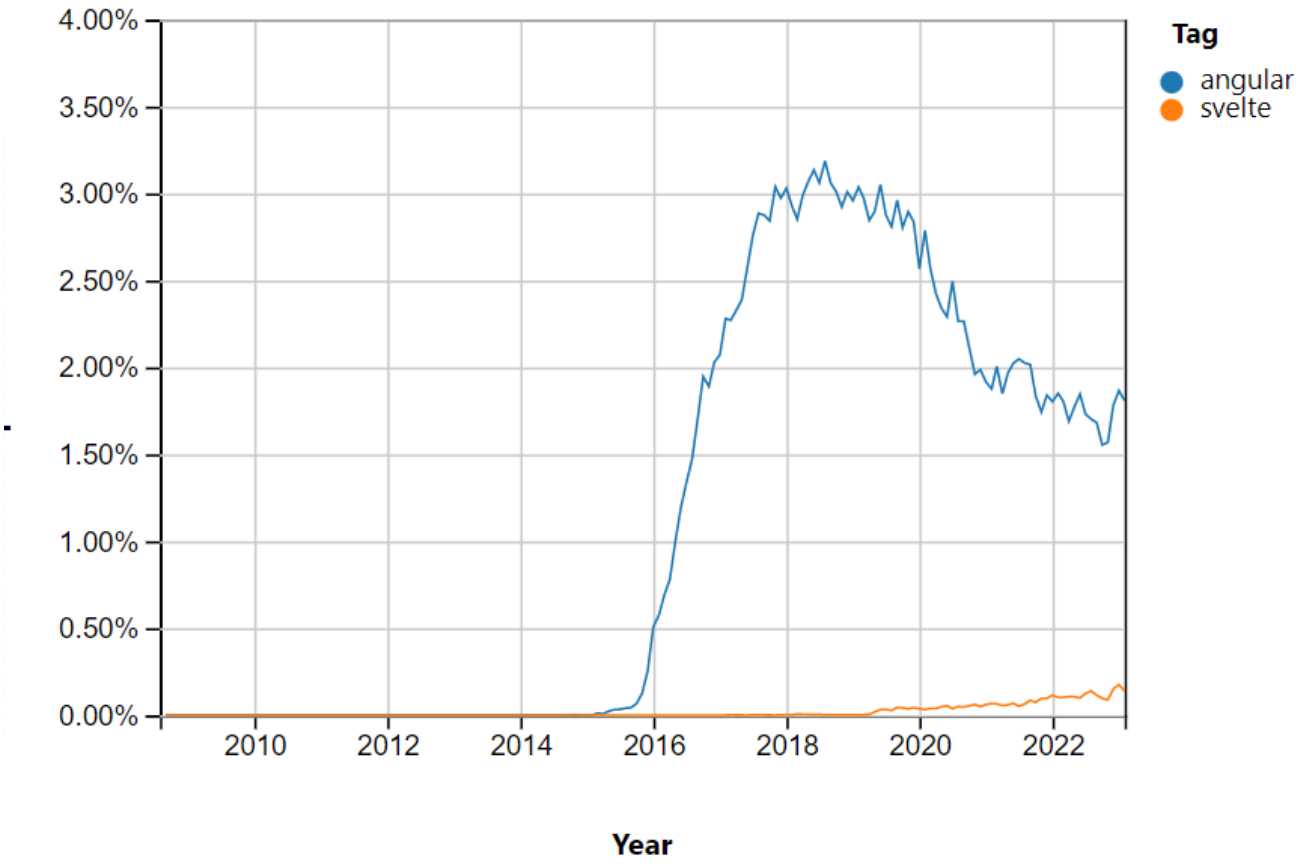
Framework	Community Support
Angular	Large, well-established community with numerous resources
Svelte	Smaller community, but growing with devoted followers

As shown, Angular boasts a larger, more well-established community than Svelte, providing access to more resources and support. However, Svelte's community is expanding and has a dedicated following, ensuring that resources are still available for developers who adopt the framework.

Stack Overflow Trends

One method of gauging a framework's popularity and community support is by examining Stack Overflow Trends. This tool tracks the number of questions asked on Stack Overflow for a specific tag over time. Let's compare the Stack Overflow Trends for Angular and Svelte:

Angular vs. Svelte Stack Overflow Trends



As the graph illustrates, Angular has been a popular framework for many years and remains in use by a vast number of developers. Svelte, as a newer framework, has gained popularity in recent years but still has a smaller community compared to Angular.

Templating with Angular

Angular employs a template-based method, where HTML templates are utilized to define an application's view. The template syntax is expanded with directives, which are unique markers in the template that instruct Angular to do something with the DOM. One of Angular's templating system's strengths is its ability to create reusable templates.

Creating Reusable Templates in Angular

Angular offers several methods for making reusable templates including:

1. ng-content
The ng-content directive is employed to project a component's content into its parent component. This results in more adaptable and reusable components, as the content can be tailored by the parent component. Here's an example:

```
<!-- Parent Component -->
<app-custom-component>
   <h1>Custom Heading</h1>
   <p>Custom Content</p>
</app-custom-component>
<!-- Custom Component -->
<div>
   <h2>Default Heading</h2>
   <ng-content></ng-content>
</div>
```

In this case, the content of the <app-custom-component> element is projected into the <ng-content> element in the custom component, allowing the parent component to customize the content.

2. ng-container
The ng-container directive is utilized to group elements and apply directives to them. This can be handy when making reusable templates. Here's an example:

```
<ng-container *ngFor="let item of items">
   <h2>{{item.title}}</h2>
   <p>{{item.content}}</p>
</ng-container>
```

In this instance, the <ng-container> element groups the <h2> and <p> elements, and the *ngFor directive is applied to the container, permitting the elements to be repeated for each item in the items array.

3. ng-template
The ng-template directive is employed to define a template that can be reused in various parts of the application. Here's an example:

```
<ng-template #customTemplate>
   <h2>Custom Heading</h2>
   <p>Custom Content</p>
</ng-template>
<ng-container *ngTemplateOutlet="customTemplate"></ng-container>
```

In this case, the #customTemplate template is established using the <ng-template> element, and then the template is reused in the <ng-container> element with the *ngTemplateOutlet directive.

Templating with Svelte

Svelte uses a custom language resembling HTML, but with some differences. One of the strengths of Svelte's templating system is its simplicity and effectiveness.

Creating Reusable Templates in Svelte

Svelte offers various ways to make reusable templates including:

1. Slots
Slots are a Svelte feature that enables more flexible and reusable components. Slots allow the parent component to provide custom content to the child component. Here's an example:

```
<!-- Parent Component -->
<CustomComponent>
   <h1 slot="heading">Custom Heading</h1>
   <p>Custom Content</p>
</CustomComponent>
<!-- Custom Component -->
<div>
   <slot name="heading">
      <h2>Default Heading</h2>
   </slot>
   <slot></slot>
</div>
```

In this instance, the <h1> element is inserted into the <slot name="heading"> element in the custom component. If there is no <h1> element supplied, the default heading will be shown instead.

2. 'Each' Blocks
'Each' blocks in Svelte make it simple to loop through arrays. Here's an example:

```
<!-- Parent Component -->
<CustomComponent items={items}/>
<!-- Custom Component -->
<ul>
   {#each items as item}
   <li>
      <h2>{item.title}</h2>
      <p>{item.content}</p>
   </li>
   {/each}
</ul>
```

In this case, the #each block is employed to go through the items array and display a list item for every element in the array.

State Management in Angular

Angular offers a variety of state management options, including services, subjects, Redux, and NgRX.

Services

Services are a popular method for managing state in Angular. They are classes that can be injected into components, allowing them to share data and logic. For example:

```
// Service
@Injectable()
export class DataService {
  private data: string;

  setData(data: string): void {
    this.data = data;
  }

  getData(): string {
    return this.data;
  }
}

// Component
export class AppComponent {
  constructor(private dataService: DataService) {}

  setData(data: string): void {
    this.dataService.setData(data);
  }

  getData(): string {
    return this.dataService.getData();
  }
}
```

In this case, the DataService is inserted into the AppComponent, and the setData and getData methods are employed to store and retrieve data.

Subjects

Subjects are a kind of observable that can be utilized for state management in Angular. They function similarly to services but can emit multiple values over time. Here's an example:

```
// Service
@Injectable()
export class DataService {
  private dataSubject = new BehaviorSubject<string>(null);

  setData(data: string): void {
    this.dataSubject.next(data);
  }

  getData(): Observable<string> {
    return this.dataSubject.asObservable();
  }
}

// Component
export class AppComponent {
  constructor(private dataService: DataService) {}

  setData(data: string): void {
    this.dataService.setData(data);
  }

  getData(): Observable<string> {
    return this.dataService.getData();
  }
}
```

In this instance, the DataService uses a BehaviorSubject to emit multiple values over time. The getData method returns an observable that can be subscribed to within the component.

Redux and NgRX

Redux is a widely-used state management library often paired with Angular. NgRX is a library that offers tools for implementing Redux in Angular applications. Here's an example:

```
// Reducer
export function dataReducer(state = '', action: any) {
  switch (action.type) {
    case 'SET_DATA':
      return action.payload;
    default:
      return state;
  }
}

// Action
export const setData = createAction('[Data] Set Data', props<{data: string}>());

// Component
export class AppComponent {
  data$: Observable<string>;

  constructor(private store: Store<{data: string}>) {
    this.data$ = store.select('data');
  }

  setData(data: string): void {
    this.store.dispatch(setData({data}));
  }
}
```

In this example, the dataReducer determines how the state is updated when an action is dispatched. The setData action is dispatched to modify the state. The AppComponent uses the store to select the data and dispatch the setData action.

State Management in Svelte

Svelte makes state management straightforward with reactive variables.

Reactive Variables

Reactive variables are a type of observable used for state management in Svelte. They can be created using the $ prefix. Here's an example:

```
<!-- Component -->
<script>
   import { writable } from 'svelte/store';

   const data = writable('');

   function setData(newData) {
   data.set(newData);
   }
</script>
<input bind:value={$data}>
<button on:click={() => setData($data)}>Set Data</button>
<p>{$data}</p>
```

In this example, we use a writable store to create a reactive variable called 'data.' The 'setData' function sets the value of 'data.' The 'bind:value' directive binds the input value to 'data,' and the '{$data}' syntax displays the value of 'data' in the paragraph element.

Comparison of State Management in Angular & Svelte

Both Angular and Svelte offer state management options, but they differ in their implementation details. Angular has several state management options, including services, subjects, Redux, and NgRX. Services and subjects are built into Angular, providing a straightforward way to manage state. Redux and NgRX, although more complex, offer greater control and flexibility.

In contrast, Svelte simplifies state management with reactive variables. These variables are easy to use and require less setup than Angular's state management options. However, as the application grows in complexity, Svelte's approach may become more challenging to manage.

Dependency Injection in Angular

Dependency injection (DI) is a design pattern that helps components depend on services provided to them instead of creating their own services. Angular has built-in support for DI, making it easy to use and maintain.

Injecting Services

Services are a common dependency in Angular. To inject a service, simply include it in the component's constructor. Here's an example:

```
// Service
@Injectable()
export class DataService {
  getData(): string {
    return 'Some Data';
  }
}

// Component
export class AppComponent {
  constructor(private dataService: DataService) {}

  getData(): string {
    return this.dataService.getData();
  }
}
```

In this example, DataService is injected into AppComponent. The getData method in AppComponent uses the injected DataService to get the data.

Providing Services

Angular uses a hierarchical injector to provide services to components. Services can be provided at various levels, including component, module, or app level. In this example, DataService is provided at the module level by including it in the providers array in AppModule, making it available to all components in the app.

```
// Service
@Injectable()
export class DataService {
  getData(): string {
    return 'Some Data';
  }
}

// Module
@NgModule({
  providers: [DataService]
})
export class AppModule {}

// Component
export class AppComponent {
  constructor(private dataService: DataService) {}

  getData(): string {
    return this.dataService.getData();
  }
}
```

Dependency Injection in Svelte

Svelte lacks built-in support for dependency injection, but third-party libraries can be used to achieve it.

Svelte-IoC

Svelte-IoC is a library that provides a simple way to implement dependency injection in Svelte. In this example, DataService is defined as a class and exported from a separate file. The inject function from Svelte-IoC is used to inject DataService into the component. The getData function uses the injected DataService to get the data.

```
<!-- Service -->
<script>
  export default class DataService {
    getData() {
      return 'Some Data';
    }
  }
</script>

<!-- Component -->
<script>
  import { inject } from 'svelte-ioc';
  import DataService from './data-service.js';

  const dataService = inject(DataService);

  function getData() {
    return dataService.getData();
  }
</script>

<p>{getData()}</p>
```

Svelte-Sugar

Svelte-Sugar is another library that offers a simple way to implement dependency injection in Svelte. In this example, DataService is defined as a class and exported from a separate file. The createInjector function from Svelte-Sugar is used to create an injector and provide the DataService. The injector.get method is used to get an instance of the DataService in the component. The getData function uses the injected DataService to get the data.

```
<!-- Service -->
<script>
  export default class DataService {
    getData() {
      return 'Some Data';
    }
  }
</script>

<!-- Component -->
<script context="module">
  import { createInjector } from 'svelte-sugar';
  import DataService from './data-service.js';

  const injector = createInjector();
  injector.provide(DataService);
</script>

<script>
  import DataService from './data-service.js';

  const dataService = injector.get(DataService);

function getData() {
return dataService.getData();
}
</script>

<p>{getData()}</p>
```

Comparison of Dependency Injection in Angular & Svelte

Angular provides built-in support for dependency injection, making it easy to use and maintain. Services can be injected into components, and they can be provided at different levels. In contrast, Svelte does not have built-in support for dependency injection but can achieve it using third-party libraries. Svelte-IoC and Svelte-Sugar both offer a straightforward way to implement dependency injection in Svelte, but they require more setup than Angular's built-in support.

Routing in Angular

Angular offers a robust routing system enabling seamless navigation between app views. RouterModule and Routes modules help achieve routing in Angular.

Setting Up Routes

Routes are defined as an array of objects in the Routes module, with each object representing a route with a path and a component. For example, the routes array defines three routes: home, about, and contact. The RouterModule is imported and configured with the routes in AppRoutingModule.

```
// Routes
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent }
];

// RouterModule
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Route Guards

Route guards manage access to specific routes, preventing unauthorized users from accessing them or performing actions before a route is accessed. For instance, AuthGuard checks whether a user is logged in before allowing access to the contact route. The canActivate method returns true if the user is logged in or false if not, redirecting them to the login page.

```
// Route Guard
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (this.authService.isLoggedIn()) {
      return true;
    } else {
      this.router.navigate(['/login']);
      return false;
    }
  }
}

// Routes
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent, canActivate: [AuthGuard] }
];

// RouterModule
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Route Resolvers

Route resolvers prefetch data before displaying a route, enhancing performance and ensuring data is available when needed. For example, DataResolver prefetches data before showing the contact route. The resolve property is set to an object containing the data property and the DataResolver.

```
// Route Resolver
@Injectable()
export class DataResolver implements Resolve<string> {
  constructor(private dataService: DataService) {}

  resolve(): Observable<string> {
    return this.dataService.getData();
  }
}

// Routes
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent, canActivate: [AuthGuard],
resolve: { data: DataResolver } }
];

// RouterModule
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Routing in Svelte

Svelte lacks built-in routing support but third-party libraries can help achieve this.

Svelte-SPA-Router

Svelte-SPA-Router is a library offering a simple way to implement routing in Svelte. For example, the Router component from Svelte-SPA-Router defines the routes, with the routes object specifying the routes and their corresponding components.

```
<!-- App -->
<script>
  import Router from 'svelte-spa-router';

  import Home from './pages/Home.svelte';
  import About from './pages/About.svelte';
  import Contact from './pages/Contact.svelte';

  const routes = {
    '/': Home,
    '/about': About,
    '/contact': Contact
  };
</script>

<Router {routes} />
```

Page.js

Page.js is another library that simplifies routing in Svelte. For example, the page library is used to define the routes with the page function specifying each route and its corresponding component.

```
<!-- App -->
<script>
  import page from 'page';
  import Home from './pages/Home.svelte';
  import About from './pages/About.svelte';
  import Contact from './pages/Contact.svelte';

  page('/', () => {
    new Home({ target: document.body });
  });
  page('/about', () => {
    new About({ target: document.body });
  });
  page('/contact', () => {
    new Contact({ target: document.body });
  });

  page.start();
</script>
```

Comparison of Routing in Angular & Svelte

Angular's built-in routing support makes it user-friendly and easy to maintain. Route guards and resolvers control access to routes and prefetch data before displaying them.

In contrast, Svelte requires third-party libraries like Svelte-SPA-Router and Page.js for routing. These libraries offer a simple way to implement routing in Svelte but may not provide the same level of control and flexibility as Angular's built-in system.

Animations in Angular

Angular offers an impressive animation system for declarative animations on components and elements using the @angular/animations module.

Creating Animations

In Angular, animations are created with the @Component decorator and the @animate decorator. For example:

```
// Component
@Component({
  selector: 'app-box',
  templateUrl: './box.component.html',
  styleUrls: ['./box.component.css'],
  animations: [
    trigger('boxState', [
      state('inactive', style({
        backgroundColor: 'blue'
      })),
      state('active', style({
        backgroundColor: 'red'
      })),
      transition('inactive => active', animate('100ms ease-in')),
      transition('active => inactive', animate('100ms ease-out'))
    ])
  ]
})
export class BoxComponent {
  state = 'inactive';

  toggleState() {
    this.state = this.state === 'inactive' ? 'active' : 'inactive';
  }
}
```

Here, the @Component decorator has an animations property that sets the boxState trigger. This trigger establishes two states (inactive and active) and two transitions between them. The BoxComponent's toggleState method changes the box's state.

Applying Animations

In Angular, animations can be incorporated into a component or element's template. For example:

```
<!-- Template -->
<div [@boxState]="state" (click)="toggleState()">Box</div>
```

This example demonstrates the [@boxState] binding, applying the boxState animation to the div element. The (click) binding activates the toggleState method in the BoxComponent.

Animations in Svelte

Svelte features a straightforward animation system for declarative animations on components and elements, achieved with the animate function.

Creating Animations

Svelte animations are created using the animate function and the use function, as shown in the example:

```
<!-- Component -->
<script>
  import { animate, use } from 'svelte/animate';

  let state = 'inactive';

  const boxState = use({
    inactive: {
      backgroundColor: 'blue'
    },
    active: {
      backgroundColor: 'red'
    }
  });

  function toggleState() {
    state = state === 'inactive' ? 'active' : 'inactive';
  }
</script>

<!-- Template -->
<div style="{animate(boxState[state], { duration: 100 })}"
on:click="{toggleState}">Box</div>
```

In this case, the animate function applies the animation to the div element. The use function defines the boxState animation, while the toggleState function alters the box's state.

Comparison of Animation with Angular & Svelte

Angular's powerful animation system enables declarative animations on components and elements. Using the @angular/animations module, animations are defined and included in a component or element's template. On the other hand, Svelte offers a simpler approach to animation using the animate function.

Angular's animation system is more robust and offers more control over the animations. Svelte's approach is more straightforward and easier to set up, but it might not provide the same level of control and flexibility as Angular's animation system.

Testing in Angular

Angular offers a thorough testing framework that enables testing of components, services, and other application elements. The modules @angular/core/testing and @angular/platform-browser/testing are used for testing in Angular.

Unit Testing

To perform unit testing in Angular, the TestBed and ComponentFixture classes are utilized. Here's a sample:

```
// Component
@Component({
  selector: 'app-box',
  templateUrl: './box.component.html',
  styleUrls: ['./box.component.css']
})
export class BoxComponent {
  state = 'inactive';

  toggleState() {
    this.state = this.state === 'inactive' ? 'active' : 'inactive';
  }
}

// Unit Test
describe('BoxComponent', () => {
  let component: BoxComponent;
  let fixture: ComponentFixture<BoxComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BoxComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BoxComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should toggle state', () => {
    const initialState = component.state;
    component.toggleState();
    expect(component.state).not.toEqual(initialState);
  });
});
```

In this example, BoxComponent is a straightforward component with a state property and a toggleState method. The unit test employs the TestBed class to set up the testing module and the ComponentFixture class to create and manage the component.

Integration Testing

Angular's integration testing is done using the TestBed and By classes. Here's a sample:

```
// Component
@Component({
  selector: 'app-box',
  templateUrl: './box.component.html',
  styleUrls: ['./box.component.css']
})
export class BoxComponent {
  state = 'inactive';

  toggleState() {
    this.state = this.state === 'inactive' ? 'active' : 'inactive';
  }
}

// Integration Test
describe('BoxComponent', () => {
  let component: BoxComponent;
  let fixture: ComponentFixture<BoxComponent>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [ BoxComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(BoxComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should toggle state on click', () => {
    const boxElement = fixture.debugElement.query(By.css('div'));
    const initialState = component.state;
    boxElement.triggerEventHandler('click', null);
    fixture.detectChanges();
    expect(component.state).not.toEqual(initialState);
  });
});
```

In this example, BoxComponent is a basic component with a state property and a toggleState method. The integration test employs the TestBed class to arrange the testing module and the By class to locate the div element.

Testing in Svelte

Svelte offers a straightforward testing framework that enables testing of components and other application elements. The libraries @testing-library/svelte and @testing-library/dom are used for testing in Svelte.

Unit Testing

In Svelte, unit testing is done using the render and fireEvent functions. Here's a sample:

```
<!-- Component -->
<script>
  export let state = 'inactive';

  export function toggleState() {
    state = state === 'inactive' ? 'active' : 'inactive';
  }
</script>

<div on:click="{toggleState}" style="background-color: {state};">Box</div>

<!-- Unit Test -->
<script>
  import { render, fireEvent } from '@testing-library/svelte';
  import Box from './Box.svelte';

  describe('Box', () => {
    it('should render', () => {
      const { getByText } = render(Box, { props: { state: 'inactive' } });
      expect(getByText('Box')).toBeInTheDocument();
    });

    it('should toggle state', async () => {
      const { getByText } = render(Box, { props: { state: 'inactive' } });
      const boxElement = getByText('Box');
      await fireEvent.click(boxElement);
      expect(boxElement).toHaveStyle('background-color: red');
    });
  });
</script>
```

In this example, Box is a simple component with a state property and a toggleState method. The unit test utilizes the render function from @testing-library/svelte to display the component and the fireEvent function from @testing-library/dom to simulate a click event.

Integration Testing

Integration testing in Svelte is carried out using the render and fireEvent functions. Here's an example:

```
<!-- Component -->
<script>
  export let state = 'inactive';

  export function toggleState() {
    state = state === 'inactive' ? 'active' : 'inactive';
  }
</script>

<div on:click="{toggleState}" style="background-color: {state};">Box</div>

<!-- Integration Test -->
<script>
  import { render, fireEvent } from '@testing-library/svelte';
  import Box from './Box.svelte';

  describe('Box', () => {
    it('should toggle state on click', async () => {
      const { getByText } = render(Box, { props: { state: 'inactive' } });
      const boxElement = getByText('Box');
      await fireEvent.click(boxElement);
      expect(boxElement).toHaveStyle('background-color: red');
    });
  });
</script>
```

In this example, Box is a basic component with a state property and a toggleState method. The integration test employs the render function from @testing-library/svelte to display the component and the fireEvent function from @testing-library/dom to simulate a click event.

Comparison of Testing in Angular & Svelte

Angular offers a comprehensive testing framework that enables testing of components, services, and other application elements. The modules @angular/core/testing and @angular/platform-browser/testing are used for testing in Angular. Unit testing in Angular is performed using the TestBed and ComponentFixture classes, while integration testing is achieved using the TestBed and By classes.

In contrast, Svelte provides a simpler testing framework that allows testing of components and other application elements. The libraries @testing-library/svelte and @testing-library/dom are used for testing in Svelte. Both unit and integration testing in Svelte are conducted using the render and fireEvent functions.

In summary, Angular provides a more extensive testing framework, which may be suitable for more complex applications, while Svelte's testing framework is simpler and more streamlined, making it a good choice for smaller projects or those with less intricate testing requirements.

Conclusion

After examining Angular and Svelte across various factors, we can see that each framework has its pros and cons. Generally, Svelte outperforms Angular in speed and size, thanks to its unique approach to app development. Svelte is also considered easier to learn due to its simpler syntax and tools, while Angular boasts a larger, well-established community with extensive documentation and support.

In terms of templates, both frameworks offer similar functionality, but Angular's template system is more powerful and supports a broader range of features. For state management, Angular provides multiple options, including services, subjects, Redux, and NgRX. In contrast, Svelte uses a simple reactive system.

Dependency injection plays a crucial role in building complex apps, and Angular delivers a robust system for managing dependencies. Svelte, however, depends on third-party libraries for dependency injection.

Routing is vital for single-page applications, and Angular offers a strong routing system with support for route guards and resolvers. Although Svelte's routing is simpler, it can still be employed to create complex applications.

Both Angular and Svelte come with animation systems that enable declarative animation of components and elements. Angular's animation system is more comprehensive, while Svelte's is easier to use.

Regarding testing, Angular features an extensive testing framework for components, services, and other application parts. On the other hand, Svelte offers a more straightforward testing framework focused on components and other application elements.

In conclusion, both Angular and Svelte are potent frameworks for developing intricate web applications. Angular is a better fit for large-scale projects requiring advanced tooling and comprehensive documentation, whereas Svelte is more suitable for smaller applications prioritizing high performance and a simpler development experience. Ultimately, the decision between Angular and Svelte will depend on the specific needs of the application and the preferences of the development team.

Mirzet Murtić

Mirzet is a seasoned frontend developer with a decade of experience, including seven years specialized in Angular. Before his work in software engineering, he had a colorful early career as a DJ, music producer, and label manager.

Mirzet's free time is spent with his kids and on hobby projects around the house. He's also a passionate amateur cynologist who devotes a lot of attention to training his dogs.