Angular Universal

```
ng add @nguniversal/express-engine
```

After this command three scripts are added to package.json

1. Dev Server
2. Prod build with Server
3. Pre-render (pre-render having universal features but without a server).

BrowserModule is for frontend

ServerModule is for server

When you do **build:ssr**, it generates frontend files and backend server(main.js) for you to run and host a server.

Angular Universal Prerendering

```
npm run prerender
```

Check `index.html`, will generate and keep rendered html. It gives exact page you get from server side but already rendererd.

Whichever routing page you will be having, it will create a directory and put it in it.

Pre-render will not create page with dynamic path param instead it works with angular routing when you traverse via angular context but if you refresh the page, you will not find the page with dynamic param.

    ng run angular-universal-course:prerender --routes /courses/01

    This will create an index.html.

You can also specify routes by defining it in a file and specifying that file.

routes.txt

/courses/01

/courses/02

```
ng run angular-universal-course:prerender --routesFile routes.txt
```

App Shell

```
<div class="" *appShellRender>
      <mat-spinner></mat-spinner>
</div>

@Directive({
      selector: "[appShellRender]"
})
export class AppShellRenderDirective implements OnInit {
      constructor(@Inject(PLATFORM_ID) private platformId,
      private templateRef: TemplateRef<any>,
      private viewContainer: ViewContainerRef) {}

      ngOnInit() {
            if(isPlatformServer(this.platformId)) {
                  this.viewContainer.createEmbeddedView(this.templateRef);
            } else {
                  this.viewContainer.clear();
            }
      }
}
```

Manually creating appshell. Only work when rendered by Server.

You can also use this above example to restrict content from loading on server side and load only on client side. Create opposite of it to do so.

Angular Universal State Transfer API

If you are using an SSR or prerendered application strategy then the process is roughly this:

1. Prerender or render application on the server
2. The browser fetches the rendered HTML and CSS and displays the "static" application
3. The browser fetches, parses, interprets and executes JavaScript
4. The Angular application is bootstrapped, replacing the entire DOM tree with the new "running" application
5. The application is initialized, often fetching data from a remote server or API
6. The user interacts with the application

There are two problems in this scenario:

1. DOM hydration is currently replacing the entire tree of nodes and re-painting the application
2. The application is fetching the data it, in theory, already had and was displayed to the user due to the SSR or prerendered site strategy

The first issue is something that is not currently solved in Angular, and quite frankly, is a complex challenge. However, the second issue is currently solved in Angular (version 9 as of this writing) using `TransferState.`

Transfer state is a strategy where we:

1. Fetch the data required to render the full "static" application using either the SSR or prerendering strategies
2. Serialize the data, and send the data with the initial document (HTML) response
3. Parse the serialized data at runtime when the application is initialized, avoiding a redundant fetch of the data.

Technically, the data is serialized via JSON.stringify() and parsed via JSON.parse(). Generally, we don't need to worry about that, however, as that is performed for us by the TransferState service in Angular.

If your application requires transferring state from the server to the client that does not conform to these requirements then you will need to implement a strategy for converting the data, both to, and from, what I'm calling "compliant" data. My specific use case is serializing objects from Firestore that contain DocumentReference classes.

Lets take an example of Route Resolver:

@Injectable()

```
export class CourseResolver implements Resolve<Course> {

    constructor(
        private coursesService: CoursesService,
        private transferState:TransferState,
        @Inject(PLATFORM_ID) private platformId) {

    }

    resolve(route: ActivatedRouteSnapshot,
            state: RouterStateSnapshot): Observable<Course> {

        const courseId = route.params['id'];

        const COURSE_KEY = makeStateKey<Course>("courseKey-" + courseId);

        if (this.transferState.hasKey(COURSE_KEY)) {

            const course = this.transferState.get(COURSE_KEY, null);

            this.transferState.remove(COURSE_KEY);

            return of(course);
        }
        else {
            return this.coursesService.findCourseById(courseId)
                .pipe(
                    first(),
                    tap(course => {
                        if (isPlatformServer(this.platformId)) {
                            this.transferState.set(COURSE_KEY, course);
                        }
                    })
                );
        }
    }

}
```

app.module.ts

```
@NgModule({
      imports: [
            BrowserTransferStateModule
      ]
})
```

app.module.ts

```
@NgModule({
      imports: [
            ServerTransferStateModule
      ]
})
```

Check at the bottom of the rendered file in browser, you will find a script tag with data. This is how it transfers state(data) to client.