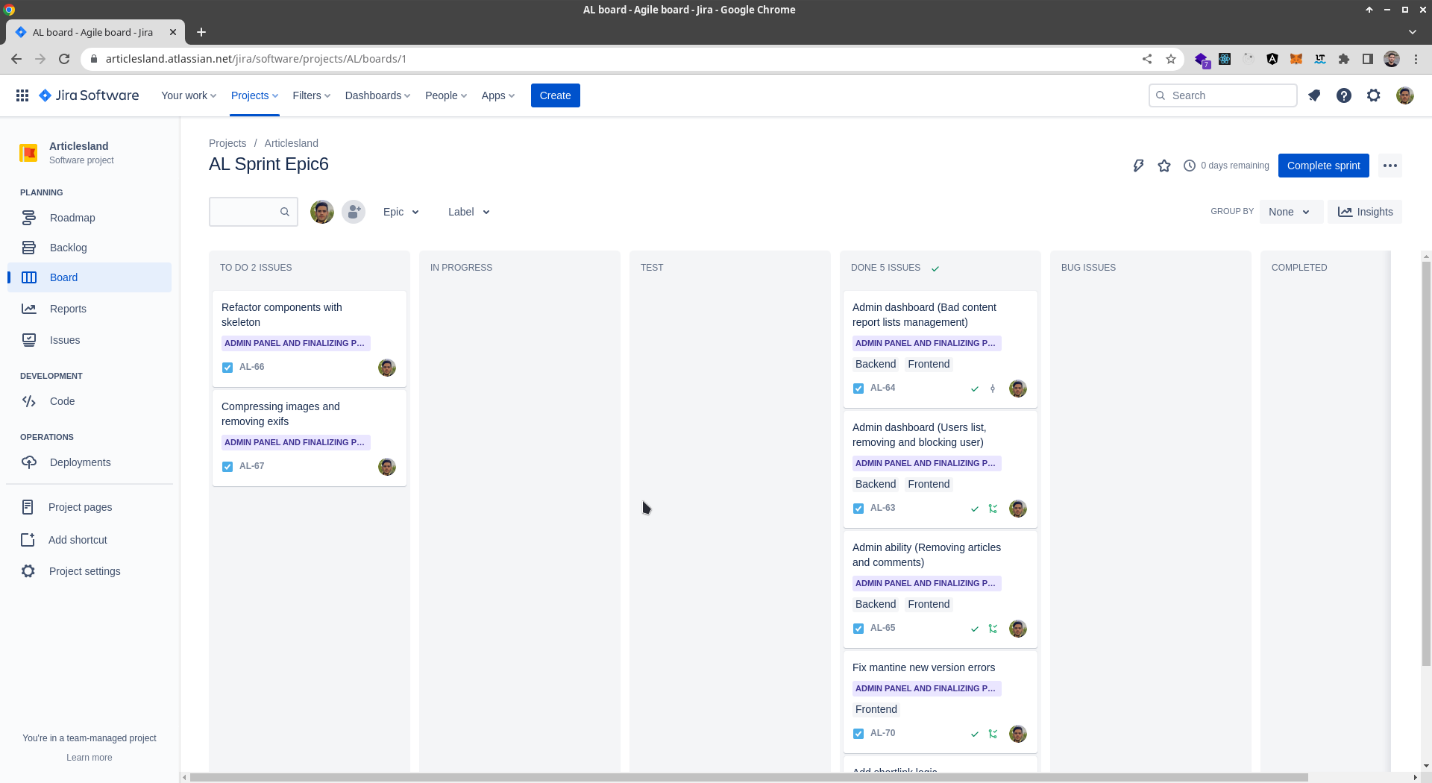
در این بخش به روش پیاده سازی پروژه بیان شده می پردازیم. ابتدا باید به بررسی نیازمندی ها و جوانب پروژه بپردازیم و سپس راه حلی برای الزامات مورد نظر پیدا کنیم. این امر در انتخاب تکنولوژی ها و نرم افزار های مورد نیاز پیاده سازی نقش بسزایی ایفا می کند.

همانطور که در سند SRS مطرح شد ابتدا متدلوژی توسعه نرم افزار را مشخص می کنیم. به عنوان یک برنامه نویس Full Stack که وظیفه همزمان اپلیکیشن سمت کاربر و سرور را بر عهده دارم، طبق تجربه به این نتیجه رسیده ام که متدلوژی Agile استفاده کنم. علت این موضوع نیز چابکی در توسعه نرم افزار است اما چابکی به چه معناست؟ دقیقا همانکاری یک برنامه نویس سطح بالای Full Stack انجام میدهد، یعنی اینکه بتواند به راحتی بین تکنولوژی های مختلف و ابعاد مختلف پروژه حرکت کند و پیاده سازی های لازم را انجام دهد اما این موضوع کافی نیست بلکه باید یک چارچوب کاری نیز برای خود انتخاب کنیم. این چارچوب می تواند scrum باشد. Scrum درواقع یک چارچوب است که توسعه نرم افزار را تحت قالب یکسری تعاریف و مفاهیم استاندارد ارائه می کند. برای توضیح این چارچوب ابتدا باید نرم افزاری که بر اساس این چارچوب عمل می کند را مشخص کنیم و برروی آن توضیحاتی ارائه دهم. نرم افزار هایی تحت وب نظیر Jira ، Trello و دیگر نرم افزار ها وجود دارند که این چارچوب ، تعاریف و مفاهیم را در قالب یکسری امکانات به ما ارائه می دهد. بنده در این پروژه از Jira استفاده کردم زیرا یک Kanban board استاندارد برای پروژه در نظر میگیرد که می توانیم وظایف ها (یا بهتر است بگوییم تسک ها) را در قالب یکسری کارت برروی ستون های این نرم افزار نمایش دهیم. شکل صفحه بعد خروجی مورد نظر را مشاهده می کنید:

بنده از نرم افزار تحت وب Jira استفاده کرده ام.



قبل از توضیح شکل باید مفهوم تسک یا Task را توضیح دهیم و اینکه تسک ها از کجا می آیند.

تسک ها وظایفی هستند که برنامه نویس باید آن ها را انجام دهد و هر تسک به شکل اجزای کوچکی از نیازمندی ها در تخته قرار میگیرد. همانطور که گفته شد تسک های قسمت کوچکی از نیازمندی ها هستند، به این دلیل به وجود می آیند که تقسیم و حل مسائل یا پیاده سازی های سامانه برای برنامه نویس شفاف و راحت تر شود.

همانطور که در شکل مشاهده میکنید تسک هایی برروی تخته مورد نظر قرار گرفته است و این ظاهر بسیار ساده Kanban board است. همچنین ستون هایی وجود دارد که روند کاری و حالت به روز یک تسک را نمایش می دهد و می توانیم متوجه شویم چه کاری انجام شده است، در حال انجام است، و انجام نشده است. ترتیب روال توسعه از چپ به راست است یعنی ابتدا تسک هایی را در قسمت To do یا انجام نشده ها قرار میدهیم و آن تسکی که اولویت بیشتری دارد در ابتدای ستون قرار خواهد گرفت. برنامه نویس یک تسک را انتخاب میکند، به خود انتساب می دهد تا هرکسی که در پروژه مشغول کار است بداند چه کاری را چه کسی در حال انجام است (اصل Social coding) و به ستون In progress یا در حال انجام انتقال می دهد. پس از پیاده سازی کد های مورد نظر ، کارت تسک را به ستون تست منتقل می کند تا میزان درستی پیاده سازی آن تسک مشخص شود و این میزان وابسته به هم پوشانی الزامات خواسته شده در تسک یا نیازمندی مورد نظر است. امکان دارد این تست توسط خود برنامه نویس انجام شود یا تیم جداگانه ای برای این کار مدنظر گرفته شوند ، اما در این پروژه بنده به تنهایی کار تست را انجام خواهم داد. در نظر داشته باشید که تست انواع متفاوتی دارد، ممکن است به شکل کد پیاده سازی شود مثل روش های TDD و BDD، یا حتی امکان دارد به صورت اجرا و تست گرفتن از خروجی نرم افزار باشد که بنده روش دوم را در این پروژه به کار گرفتم هرچند برای قسمت های مهمی از کد از روش های BDD و TDD استفاده کردم. اگر تست های مورد نظر به خوبی قبول شدند، تسک را به ستون انجام شده یا Done هدایت می کنیم در غیر این صورت به ستون Bug issues یا دارای اشکال، منتقل می کنیم و البته می توانیم روی کارت تسک کلیک کنیم تا گزینه هایی مثل نوشتن نظر را مشاهده کنیم. باید حتما قسمتی که دارای مشکل است داخل نظر نوشته شود تا توسط برنامه نویس رفع شود. این نمایی ساده و توضیحاتی کامل برای Kanban board بود.

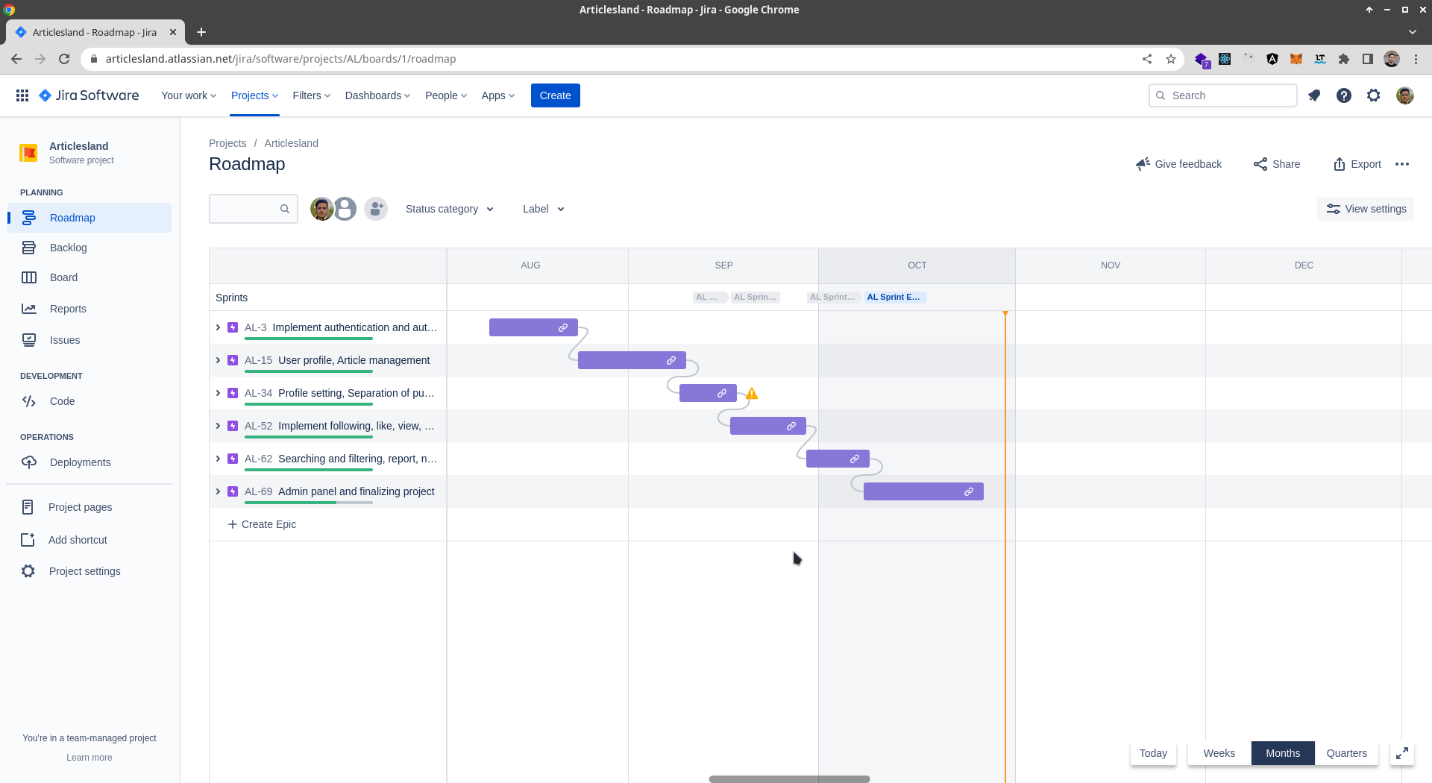
اما مفهومی وجود دارد به اسم Epic داریم، Epic درواقع User Story یا داستان کاربری است که در تخته جداگانه ای به اسم نقشه مسیر یا roadmap قرار می دهیم. هر epic شامل یک روال از عملکرد کاربر در سامانه است، برای مثال ایجاد و مدیریت مقالات که امکان دارد خود این موضوع شامل یک یا تعدادی یوزکیس باشد. برای مثال ایجاد و مدیریت مقالات خود به تنهایی شامل یوزکیس های زیر است:

1. ایجاد مقاله
2. حذف مقاله
3. ویرایش مقاله
4. انتشار مقاله
5. لغو انتشار مقاله

پس با انتخاب درست یوزکیس ها یا یک یوزکیس بخصوص به عنوان epic می توانیم انتزاعی خیلی خوب از روال کار به دست بیاوریم. در واقع epic تعریف دیگری هم دارد که برمبنای توصیف IBM از epic است. طبق مستندات IBM ، epic علاوه بر اینکه یک داستان کاربری است، یک MVP نیز است، درواقع MVP که مخفف Must Valuable Product است، بیان کننده شکلی از نرم افزار است که ممکن است نسبت به ابعاد اصلی برنامه بسیار کوچک باشد اما کاربردی و عملیاتی است. برای مثال فرض کنید یک مشتری از شما می خواهد وسیله ای برای او بسازید که بتواند به راحتی داخل شهر جابجا شود، چون مشتری مورد نظر اتوموبیل را وسیله نقلیه تمام و کمالی می داند و خواسته او از شما چیزی شبیه به اتوموبیل است، باید برای او یک اتوموبیل آماده کنیم. پس شروع به کار می کنیم تا خواسته او را عملی کنیم. پس از مدتی مشتری به ما سر میزند و می خواهد روال پیشرفت کار را مشاهده کند ولی با بدنه یا شاسی روبرو می شود، اما آیا این قطعه برای او مفید است؟ یا او را خشنود می سازد؟ به طور قطع خیر. MVP پاسخ شما برای جلب رضایت مشتری است که با هدف مشتری همخوانی دارد. طبق این استاندارد شما می توانید پس از گذشت مدتی به جای ایجاد شاسی، برای مشتری یک دوچرخه بسازید تا راحت نیازمندی اولیه اون برطرف شود، بعنی بتواند به راحتی در شهر جابجا شود. سپس این دوچرخه را بروز تر می کنید تا به یک موتور تبدیل شود و خب توان کمتر و بازدهی بیشتری برای مشتری خواهد داشت و مشتری علاوه بر رفع نیاز اولیه ، ویژگی های خیلی خوبی را دریافت می کند که آستانه رضایت او را بسیار افزایش می دهد. سپس دو چرخ را به چهار چرخ تبدیل می کنیم و صندلی و بدنه خیلی خوبی برای آماده می کنیم تا یک اتوموبیل خوب به مشتری تحویل دهیم، اما مشتری از ابتدای روال توسعه تا انتها همواره درحال سود بردن از سامانه بوده است پس رضایت بسیار زیادی به همراه داشته است. به نقل از IBM مطمئن باشید آنقدر جای پای شما در ایجاد سامانه محکم شده است و به خوبی روی ابعاد مسلط هستید که در صورت موافقت مشتری می توانید به اون هواپیما تحویل دهید!

پس از توصیف MVP از نقطه نظر IBM به ادامه توضیح می پردازیم.

در شکل زیر یک roadmap یا نقشه راه را مشاهده میکنید:



همانطور که مشاهده می کنید ، از یوزکیس ها یک انتزاع برمبنای User Story و MVP ، تعدادی Epic ایجاد می کنیم. تسک هایی که قبلا ایجاد کرده ایم، برمبنای هر Epic است، بعنی به ازای هر Epic یک kanban board ایجاد می کنیم و تسک های مورد نیاز آن Epic را استخراج و در ستون انجام نشده ها یا To do قرار می دهیم.

طبق چارچوب scrum یک مفهوم بسیار مهم به اسم sprint داریم. Sprint به معنای یک دوره توسعه نرم افزاری می باشد که تسک هایی را قرار است در آن دوره به پایان برسانیم. طبق توضیحات داده شده معمول است که برای هر Epic یک sprint مدنظر قرار می دهند که بنده نیز از این قاعده پیروی کرده ام. تسک هایی که به صورت پیش نویس آماده می شوند و خام هستند را backlog می گویند. این تسک ها مستقیما با بررسی نیازمندی های کاربر یا توصیفات کاربری استخراج می شوند و ممکن است قابلیت پیاده سازی آن ها ، مبهم باشد یا غیر ممکن که پس از بررسی های لازم به یک تسک قابل پیاده سازی تبدیل می شود. اما چونکه قبل از پیاده سازی ، نمودار های یوزکیس آماده شده است پس بسیار راحت تر می توانیم تسک ها را ایجاد کنیم و در backlog قرار ندهیم.

ما می توانیم در نرم افزار Jira کد هایی که روی SCM یا Source Control Manager قرار دارند را با تسک ها همگام سازی کنیم و به یکدیگر متصل کنیم.

Source Control Manager: نرم افزاری است که با امکان مدیریت کد منبع، ادغام و یکپارچگی انجام دهیم که از مهمترین این دسته از نرم افزار ها می توانیم به git اشاره کنیم. Git دارای ابزار و مفاهیم خاص خود است که روی سیستم شما یک local repository ایجاد می کند و می توانید داخل مخزن یا repository کد خود را توسعه دهید.

امکانات SCM:

* کنترل نسخه های مختلف نرم افزاری
* گزارش گیری از روند توسعه
* مجتمع کردن کد های سامانه داخل یک مسیر
* ایجاد روند های جداگانه از توسعه به اسم branch برای مختل نکردن توسعه بقیه توسعه گر ها یا اختلال در کد اصلی برنامه
* ادغام کردن روند های ایجاد شده یا همان branch ها
* کشف تعارضات حاصل از ادغام بین branch ها
* افزودن نرم افزار های دیگر که در قسمت استقرار
* تست و امنیت متمرکز شده اند روی مخزن اصلی

همچنین وب اپلیکیشن های دیگری برای افزودن ویژگی های بیشتر به این ابزار بوجود آمدند که از مشهور ترین آن ها می توان از موارد زیر نام برد.

* Github
* Gitlab
* Bitbucket

که این وب اپلیکیشن ها امکان همگام یا سینک شدن دیگر نرم افزار های حوزه ی DevOps را می دهد و البته از مهمترین ویژگی های آن مفهوم remote repository یا مخزن از راه دور است، این ویژگی به برنامه نویسان یک تیم این امکان را می دهد که روی فضای ابری این سامانه های کد های خود را بارگزاری یا بارگیری کنند و یک فضای کار اشتراکی را ایجاد کنند.

نرم افزار های دیگری وجود دارند مانند Jira که با این پلتفرم ها همگام یا سینک می شوند تا بتوانند سرویس های خود را ارائه دهند. برای مثال Jira می تواند به هر سه پلتفرم ذکر شده متصل شود و کد های آن را رصد کند تا تغییراتی که اعمال می شود را با اطلاعات خود ادغام و در نهایت برای کاربر گزارش مفیدی ارائه دهد، یکی از ویژگی های خوب Jira این است که برنامه نویس را مجاب می کند تا هر تسک را به شکل یک branch جداگانه پیاده سازی کند و در صورتی که کد را بارگزاری کرد ، ادمین مخزن بیاید و branch ایجاد شده را با branch اصلی که درواقع همان کد اصلی سامانه است، ادغام یا merge کند. این مورد باعث تمیز شدن روند پیاده سازی تسک ها و توسعه کد های روی remote repository می شود که برای این پروژه از github استفاده شده است.

تا این مرحله ، متوجه روند توسعه کد های نرم افزاری بر مبنای متدولوژی agile و چارچوب scrum شدیم. پس epic ها و تسک های مربوط به سامانه را استخراج می کنیم و بعد آن به github متصل می کنیم. حال نوبت آن است که در مورد تکنولوژی های مورد استفاده و علل استفاده از آن ها صحبت کنیم.

در این قسمت باید به ابعاد ملزومات نوشته شده داخل مستندات SRS توجه ویژه ای کرد که برای مرور یکبار دیگر در این قسمت نوشته می شود و نیازمندی ها را به شکل عملیاتی و فنی پاسخ می دهیم.

ملزومات اجرایی

* در اپلیکیشن سمت سرور پردازش سنگینی وجود نخواهد داشت اما باید سرعت پاسخ بالا باشد.
* ذخیره محتوای مقاله به صورت خودکار نباید دارای حافظه مشترک ذخیره سازی فایل با سرور اصلی داشته باشد و اگر جدا شود بهتر است زیرا سربار پردازشی ذخیره و بازیابی نیز برروی سرور اصلی کم می شود.
* اپلیکیشن سمت کاربر باید طوری پیاده سازی شود که کمترین تعداد درخواست را با اپلیکیشن سمت سرور داشته باشد، یعنی نباید سربار rendering یا نمایش صفحات با اپلیکیشن سمت سرور باشد و سرور تنها مسئولیت پاسخ گویی به API ها را دارد.
* گاهی نیاز است که اپلیکیشن سمت کاربر قبل از نمایش صفحه یک درخواست عمومی به سمت اپلیکیشن سمت سرور داشته باشد که بهتر است یک سرور مجزا که داخل شبکه داخلی اپلیکیشن اصلی سرور است ، درخواست های متداول و عمومی را بزند و پاسخ را به شکل صفحه ایجاد شده برای کاربر ارسال نماید تا سربار سمت کاربر کاهش یابد.

ملزومات ایمنی

* بهتر است طبق استاندارد های DevSecOps ، از ثبت کردن نقص های نرم افزاری استفاده کنیم ( برای مثال داخل فایل – تکنیک logging ) تا بتوانیم در اسرع وقت مشکلات را پیدا و عیب یابی کنیم.
* بهتر است برای پیاده سازی نرم افزار از تکنیک های برنامه نویسی TDD و BDD برای تست واحد و یکپارچه (unit test and integration test) استفاده کنیم.
* برای پیاده سازی بسیار مفید است که از زبان هایی استفاده کنیم که قابلیت خود استنادی یا self-documenting داشته باشند.

ملزومات امنیتی

* بهتر است طبق استاندارد های DevSecOps و OWASP Top 10 ، از پروتکل های امنیتی و جلوگیری از خطا های رایج در پیاده سازی گام های مختلف چرخه توسعه نرم افزار استفاده کنیم.
* سیستم های تداوم توسعه ، تداوم استقرار و تداوم ارائه یا CICD می تواند بسیار در ملزومات امنیتی و ایمنی کارایی داشته باشند.

در قسمت ملزومات اجرایی ذکر شده است که اپلیکیشن سمت سرور باید I/O bound باشد. دلیلش هم آن است که پردازش مستقیم برای محاسبات سیستمی نداریم و فقط باید پاسخ درخواست ها سریع داده شود. اگر ورودی همان درخواست آمده به سمت سرور و خروجی همان پاسخ داده شده از سمت سرور باشد، سیستم ما I/O bound یا I/O limited تلقی می شود.

برای پاسخ به این نیازمندی یکی از بهترین گزینه ها node.js است که در واقع یک محیط اجرایی برای زبان JavaScript تحت سیستم عامل می باشد و چون تست سیستم عامل اجرا می شود می تواند به کتاب های سیستم عاملی دسترسی داشته باشد، در نهایت به این نتیجه می رسیم که node.js می تواند از کتاب های http برای ایجاد یک وب سرور بهره ببرد. http یا Hyper Text Transfer Protocol یک پروتکل شبکه در لایه application است که برای اپلیکیشن های تحت وب مورد استفاده قرار میگیرد.

اما امروزه می دانیم که از فریم ورک یا framework ها برای توسعه اپلیکیشن های استفاده می کنند اما تعریف فریم ورک به شرح زیر است:

مجموعه ای کتابخانه ها ، ماژول ها ، Design Pattern ها هستند که به کمک آن ها سعی می کنیم یک کد استاندارد بر پایه مستندات آن فریم ورک ایجاد کنیم. داخل فریم ورک ها سعی در منسجم کردن مدل های طراحی ، معماری و Design Pattern یا الگو های طراحی شده است.

الگو های طراحی یا Design patterns ، درواقع در گام اولیه Solution pattern یا الگو های راه حل بوده اند که بسط آن بر مبنای کاربرد مورد نظر ، به شکل یک قائده کلی استفاده شده اند. درواقع الگو های راه حل یا Solution patterns یکسری الگوریتم هایی هستند که در قسمت هایی از کد های نرم افزاری به کار گرفته شده اند تا مشکلات خاصی را پاسخ دهند. اما الگو های طراحی یک رویکرد کلی تر و مدل پیشرفته و باقائده ای از الگو های راه حل هستند که می توانند توسط برنامه نویسان به شکل یک استاندارد پذیرفته شوند و کد های خود را توسعه دهند. ویژگی های مهم آن قابل بسط دادن برای ارائه راه حل های احتمالی سامانه ها است. برای مثال فرض کنید که می خواهید یک اپلیکیشن سمت سرور بنویسید، وقتی یک برنامه اجرا می شود درواقع یک پردازه یا process در سیستم عامل به وجود می آید که CPU آن را پردازش می کند. همانطور که در مورد روند طبیعی اجرای نرم افزار صحبت کردیم، ساختار کد های برنامه می تواند به گونه ای باشد که تنها یک شیء کلی از تمامی کد های نرم افزار گرفته شود و پس از صدا زدن تابع اجرای آن شیء ، برنامه مورد اجرا شود. این دیدگاه هم باعث جلوگیری از گرفتن نمونه های اضافی از کد برنامه می شود که در نهایت حافظه process مورد نظر را به طور بیهوده ای مصرف می کند هم ما از دوباره نویسی کد یا خودتکراری جلوگیری می کنیم. این رویکرد تحت عنوان Singleton Design Pattern شناخته شده است که ساختار خاص خود را دارد و امکان دارد در الگوی طراحی دیگر استفاده شود یا از انواع دیگری از الگو های طراحی استفاده کند. برای مثال اگر بخواهیم از singleton استفاده کنیم باید طبق تغریف آن از شیء گرایی استفده کنیم یا اینکه آن را می توانیم در MVC Design Pattern استفاده کنیم که مخفف Model View Controller است که در آینده به شرح آن می پردازیم.

من در این پروژه از فریم ورک Nest.js برای اپلیکیشن سمت سرور استفاده کردم که کد های بسیار استاندارد تولید می کند. این فریم ورک خیلی شبیه Angular است پس از استاندارد های الگو های طراحی Google تبعیت می کند.

در ملزومات اجرایی قسمت دوم بر مجزا بودن فضای ذخیره سازی و البته روال پردازشی ذخیره سازی سمت اپلیکیشن سمت سرور تاکید شده است. این به این معنا است که پردازه یا process جداگانه ای نیازمند هستیم زیرا JavaScript زبان single thread است یعنی نهایتا یک نخ داخل process است که در حال اجرای کد های برنامه است. حال فرض کنید که هر سه دقیقه تعداد زیادی کاربر قرار است به سمت سرور درخواست بزنند، طبیعتا در آن لحظه سرور نیاز دارد پاسخ باقی API ها که بسیار مهمتر هستند را به کاربران بدهد اما سربار سرور بسیار زیاد می شود. راه حل معماری Microservice است.

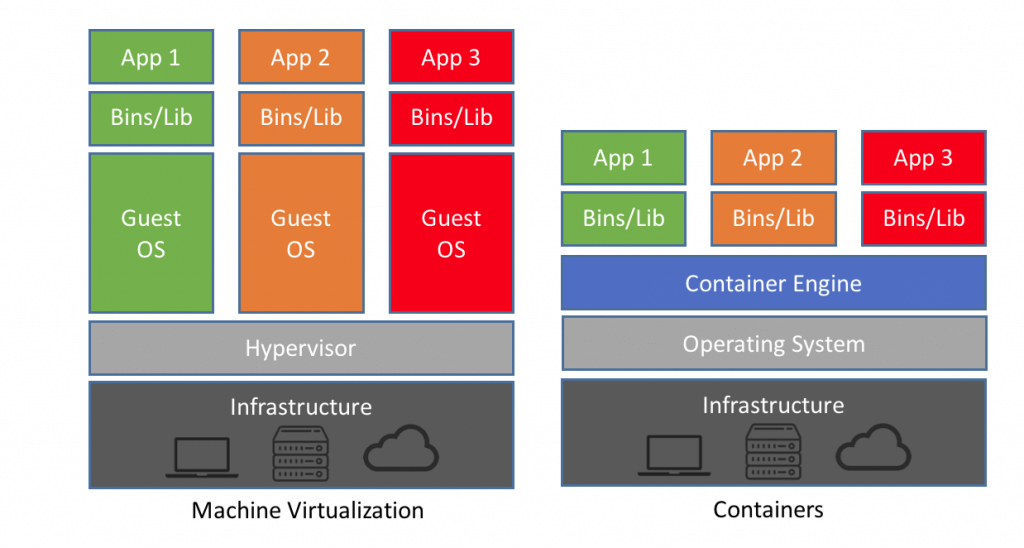
این معماری چندین سرویس متفاوت را که هرکدام به تنهایی process یا پردازه های متفاوتی هستند با نوع ارتباط خاص، به یکدیگر متصل می کند. ارتباط آن ها به شکلی است که روی یک صف یا چندین صف، درخواستی از سمت یک سرویس فرستنده روی صف گذاشته می شود تا سرویس دیگری آن سرویس را بردارد و زمانی که سرویس مقصد درخواست را دریافت کرد، پردازش لازم را انجام دهد و پاسخ را روی صف گذاشته و سپس سرویس فرستنده پاسخ را دریافت کند. درواقع بهتر است به جای درخواست و پاسخ بگوییم پیام، زیرا این دو از یک جنس هستند و اصطلاح فنی سرویس ها به کمک پیام ها با یکدیگر در ارتباط هستند. به پلتفرمی که این بین نقش جابجایی پیام ها را دارد ، message broker گفته می شود که بنده از RabbitMQ استفاده کرده ام اما انواع دیگری مثل Apache Kafka، پیام در سطح TCP و دیگر موارد نیز وجود دارد.

معمولا در این نوع معماری از container ها استفاده می شوند که توسط یک engine مدیریت می شوند. هر container یک واحد نرم افزاری است که کد ها و تمام وابستگی های آن (dependencies) بسته بندی می کند تا نرم افزار با سرعت و قابلیت اطمینان بیشتری اجرا شوند با این نگرش که محیط محاسباتی یا computing مجزایی داشته باشد. هر container از image ها نمونه گیری و اجرا می شوند. این image ها درواقع حاوی سیستم عامل لینوکس برای سبکی هرچه بیشتر و البته وابستگی های نرم افزاری مورد استفاده یک نیازمندی خاص می باشد. برای مثال وقتی می خواهم یک برنامه را با استفاده از Node.js اجرا کنم می توانم از Image لینوکسی (distribution ها می توانند متفاوت باشند) که روی آن Node.js نصب و تنظیم شده است استفاده کنیم به طوری که یک container از image نمونه بگیریم و روی آن کد های مورد نظر را کپی و برنامه را اجرا کنیم.

چونکه محیط های محاسباتی یا computing متفاوت است، پردازش جداگانه دو برنامه JavaScript بسیار سریع تر است. همچنین محل ذخیره سازی فایل ها نیز متفاوت خواهد بود.

من برای پیاده سازی این ویژگی از Docker که یک Container Engine است استفاده کرده ام ولی برای اتصال و خودکار سازی ایجاد Container ها از Docker Compose استفاده شده است.

ممکن است خوانندگان محترم منطق Hypervisor و Virtual Machine هایی که با سیستم عامل مجزا اجرا می شوند اشتباه بگیرند. توضیح این موضوع از عنوان و هدف مقاله خارج است ولی می توانید جداگانه مطالعه بفرمایید.

 نکته: یکی دیگر از مهمترین مزایای container ها ، اجرا شدن اپلیکیشن های متفاوت داخل هر نوع سیستم عاملی است که برروی آن Docker نصب شده است. این مورد باعث می شود اعضای تیم که با سیستم عامل های متفاوتی برنامه نویسی می کنند دچار مشکل نشوند حتی اپلیکیشن های مورد نظر برروی سرور ابری نیز به یک روش اجرا خواهد شد.

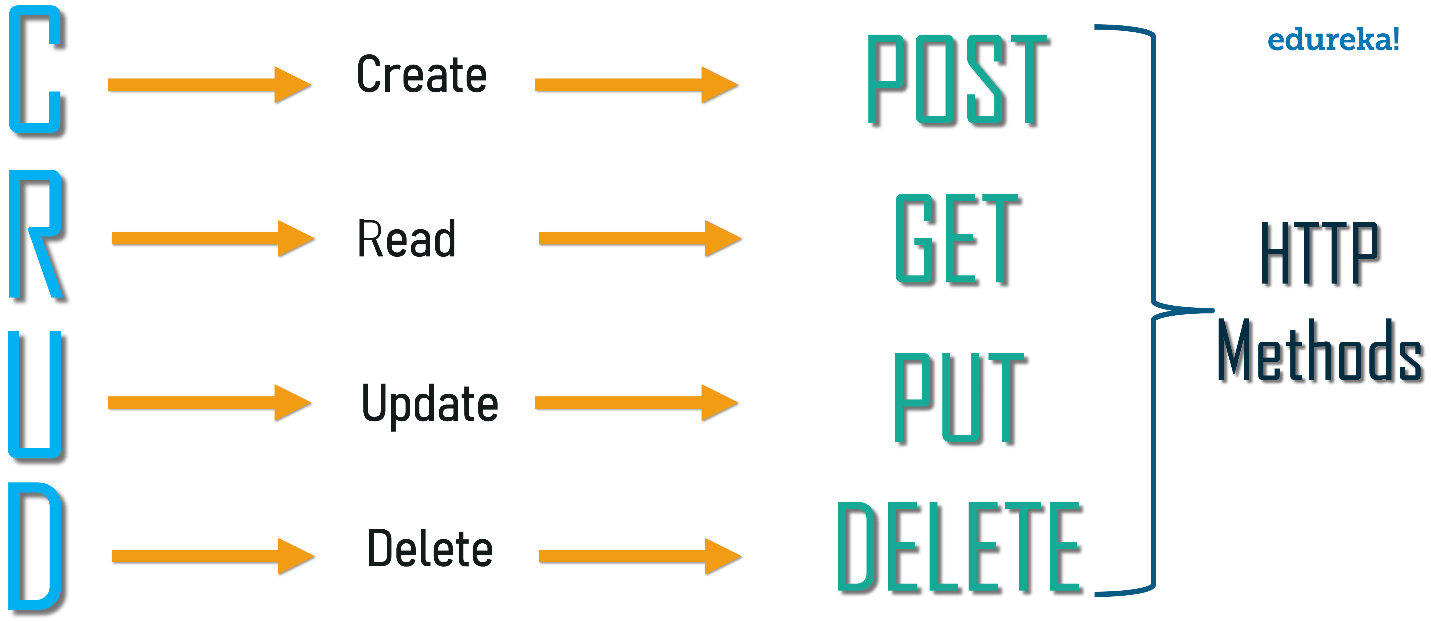
در ملزومات اجرایی قسمت سوم بر مجزا بودن عملیات rendering صفحه از اپلیکیشن سمت سرور تاکید شده است. برای حاصل شدن نتیجه مطلوب می توانیم از SPA ها استفاده کنیم.

SPA یا Single Page Application یک رویکرد و تکنولوژی نوین برای اپلیکیشن های سمت کاربر است، منطق آن بر این پایه و اساس چیده شده است که قالب اولیه صفحه وب در قالب HTML برای کاربر ارسال می شود، سپس کد های JavaScript برای rendering صفحه سمت کاربر ارسال شود، در این صورت کاربر کل اپلیکیشن سمت کاربر را یکجا یا خرد خرد دریافت می کند و با سر هم گذاشتن آن ها صفحه مورد نظر را می تواند مشاهده بکند. ولی اگر کد JavaScript ، مبهم سازی نشده باشد و البته کم حجم نشده باشد، این فرایند دارای سربار بسیاری خواهد بود، به این معناست که مبهم سازی باعث کاهش حجم متن کد مثل نام توابع و متغیر ها می شود و ساختار کد را تو در تو می کند، کم حجم سازی نیز کد های زبان انعطاف پذیر JavaScript را فشرده می کند به طوری که هم فاصله هم و خط های جدید غیر کاربردی حذف خواهند شد. پس حجم کد به شکل چشم گیری کاهش می یابد. یادمان باشد که محتوای دارای HTML و استایل های CSS داخل خود JavaScript قرار دارند و به کمک DOM داخل صفحه نمایش داده می شوند. DOM مخفف Document Object Model است یعنی این که کله بدنه و شاکله کلی HTML را به صورت شیء ببینیم و البته المنت هایی که داخل آن قرار دارند را به شیء های داخلی آن شیء اصلی مدل کنیم.

در حال حاضر سه تا از معروف ترین فریم ورک ها برای ایجاد SPA می توانیم از React.js ، Angular و Vue.js نام ببریم.

بنده در این پروژه از React.js استفاده کردم. به دلیل ساختاری که مورد پسند بنده برای توسعه اپلیکیشن بود. همچنین نسبت به دو فریم ورک دیگر دارای جامعه توسعه دهندگان و البته کتابخانه های بیشتری است.

با این منطق اپلیکیشن سمت سرور هیچ عملیاتی rendering انجام نخواهد داد بلکه API ها را به شکل REST API در معرض استفاده اپلیکیشن سمت کاربر قرار خواهد داد. REST API در واقع اشاره به شکل ، ساختار و نوعی قائده برای ایجاد درخواست به سمت اپلیکیشن سمت سرور می باشد.



این قائده به گونه ای است که یک مسیر اصلی برای یک دسته از عملیات های مربوط به یکدیگر وجود دارد ولی نوع عملیات وابسته به نوع درخواست HTTP است. برای مثال فرض کنید مسیر زیر در بر گیرنده منطق کارکرد با مقالات باشد: /post

حالا اگر بخواهیم از قائده CRUD پیروی کنیم، اگر به این مسیر درخواست GET بزنیم ، باید لیستی از مقالات را دریافت کنیم. زیرا درخواست خواندن داده ایم. اگر درخواست از نوع POST بزنیم و یک بدنه مثلا با فرمت JSON ارسال کنیم، تابعی سمت اپلیکیشن سرور اجرا خواهد شد که برحسب بدنه ارسالی، یک مقاله جدید ایجاد میکند. برای درخواست PUT و DELETE هم به ترتیب ، روال ویرایش و حذف مقاله وجود خواهد داشت.

در ملزومات اجرایی قسمت چهارم بر مجزا بودن عملیات rendering صفحه های عمومی از اپلیکیشن سمت سرور و البته render مستقیم توسط SPA تاکید شده است. برای حاصل شدن نتیجه مطلوب می توانیم از فریم ورک Next.js استفاده کنیم. در React.js بدلیل اینکه تمامی منطق و صفحات برای کاربر به صورت یکجا ارسال می شود تا مرورگر کاربر به rendering مشغول شود، می توانیم قبل از rendering یکسری توابعی داشته باشیم که به صورت مجزا اجرا می شوند. برای بیان ساده تر استراتژی های rendering را معرفی میکنیم:

* Client Side Rendering یا CSR: این مدل rendering به گونه ای است که اپلیکیشن سمت کاربر درخواست هایی به سمت اپلیکیشن سمت کاربر میزند و نتیجه را گرفته و سپس در صفحه محتویات دریافت شده را تحت قالب های خاص HTML نمایش می دهد.
* Server Side Rendering یا SSR: این مدل rendering به گونه ای است که اپلیکیشن سمت کاربر توسط یک سرور جداگانه اجرا می شود و اجزایی که نیاز به CSR دارند را مستقیما برای کاربر ارسال می کند اما صفحاتی هستند که قبل از ارسال برای مرورگر کاربر، باید داده هایی را از سرور اصلی گرفته و پس از دریافت صفحه را کامل render کرده سپس برای مرورگر کاربر ارسال کن تا فقط نمایش داده شود، یعنی سربار rendering برروی سرور مخصوص اپلیکیشن سمت کاربر قرار میگیرد.

اما فایده SSR برای زمانی است که می خواهیم یکسری درخواست را محافظت کنیم به طوری که ممکن است از یک سرویس ثانویه استفاده کنیم و نخواهیم اطلاعات درخواست را کاربر ببیند. زیرا گفتیم که در CSR تمامی فایل های JavaScript که وظیفه rendering را دارند مستقیما برای کاربر ارسال می شود پس اگر سرویس ثانویه داشته باشیم مجبور هستیم درخواست ها را از سمت کاربر بزنیم که این موضوع بسیار خطرناک هست زیرا کلید ها و اطلاعات دسترسی به API های پلتفرم ثانویه در دسترس کاربر قرار میگیرد. ما در این اپلیکیشن از این دسته از API ها را نداریم.

اما کاربر دیگری که ما استفاده می کنیم نیز جنبه حفاظتی و سرعت پردازش دارد. فرض کنید که کاربر می خواهد یک پست را دریافت و مشاهده کند. در صفحه مشاهده پست میرود و صفحه مشاهده پست درخواستی را به سرور اصلی مبنی بر دریافت اطلاعات پست میزند و سپس داده ها را میگیرد و در صفحه نمایش می دهد. حال فرض کنیم که این مقاله در حالت پیش نویس باشد و کاربر دیگری نتواند آن مقاله را ببیند. در این صورت باید صفحه 404 یا صفحه "موردی یافت نشد" برای کاربر نمایش داده شود. در این صورت باید اپلیکیشن سمت کاربر مدتی را منتظر باشد تا پاسخ 404 برگردد و سپس خودش را به صفحه "موردی یافت نشد" هدایت کند.

این مکانیزم با SSR خیلی بهتر عمل خواهد کرد. اما چرا؟

پاسخ به این شکل است که اپلیکیشن سمت کاربر به سروری متصل است که مسئولیت کنترل آن را دارد. وقتی کاربر به صفحه نمایش پست درخواست میزند، سرور اپلیکیشن سمت کاربر به اپلیکیشن اصلی سمت سرور درخواست درون شبکه ای میزد زیرا اگر روی یک زیرساخت ابری مستقر شده باشند سرعت بسیار بالایی در پاسخ شاهد خواهیم بود. پس از دریافت اطلاعات مقاله صفحه را ایجاد می کند و برای کاربر ارسال می کند. از نقطه نظر پردازش نیز وابستگی به اپلیکیشن سمت سرور اصلی وجود ندارد زیر سروری مجزا برای کنترل rendering اپلیکیشن سمت کاربر وجود دارد یعنی یک process مجزا از سرور اصلی. از طرفی فرض کنید که مقاله وجود ندارد، خیلی سریع سرور کنترل کننده اپلیکیشن سمت کاربر، صفحه "موردی یافت نشد" را برای کاربر نمایش خواهد داد. این مکانیزم عملکرد خیلی بهتری خواهد داشت.

ولی باید بدانید زمانی این مکانزیم مفید است که API مورد نظر دارای محدودیت احراز هویت نیست و یک API عمومی می باشد زیرا اپلیکیشن سمت کاربر که روی مرورگر اجرا می شود به توکن احراز هویت دسترسی دارد نه سروری که نقش کنترل کردن rendering را بر عهده دارد.

نکته: طبق مواردی که بیان شد ما باید سه سرور در حال اجرا روی زیرساختی که اپلیکیشن ها قرار است مستقر شوند خواهیم داشت. درواقع به طور دقیق پنج عدد container که دو مورد از آنها پایگاه های داده هستند. یکی پایگاه داده MariaDB برای سرور اصلی و یکی دیگر MongoDB برای سرور caching یا ذخیره محتوای مقاله به صورت اتوماتیک. یکی دیگر از container ها اپلیکیشن RabbitMQ است که قبلا توضیح داده شد و برای اتصال دو سرور سمت سرور استفاده می شود. دو container دیگر برای دو سرور دیگر استفاده خواهد شد زیرا با استفاده از شبکه مجازی داخل Docker Engine می توانند به کمک RabbitMQ به یکدیگر متصل شوند. همچنین وقتی از image های MariaDB و MongoDB برای ایجاد container ها، استفاده می کنیم دیگر درگیر نحوه نصب و پیکربندی پایگاه داده رو سیستم میزبان نمی شویم. می توانیم اپلیکیشن سمت کاربر همراه با سرور کنترل کننده را تحت قالب یک container اجرا کنیم ولی نیازی نیست زیرا نیازی به شبکه شدن با بقیه سرور ها را ندارد، برنامه نویسان دیگر نیز می توانند به راحتی با استفاده از کد منبع آن را اجرا کنند.

برای استقرار یا Deployment اپلیکیشن ها ، باید از زیرساخت های ابری IaaS استفاده کنیم که مخفف Infrastructure as a Service است. سرویس دهنگان ابری سرور هایی را تحت قالب IaaS ارائه می کنند که با SSH یا دسترسی Secure Shell می توانیم به آن متصل شویم، داکر را نصب کنیم و پس از بارگیری کردن کد های منبع از مخازن یا repository های github، آن ها را اجرا کنیم.

اما استقرار کردن اپلیکیشن ها هنوز به پایان نرسیده است. بنده از تکنیک reverse proxy و استفاده از یک وب سرور واسط، عملیات استقرار کردن نرم افزار را تکمیل کردم. Reverse proxy یک روشی است که درخواست هایی که به سمت سرور ابری و وب سرور ارسال می شود را، به سمت وب اپلیکیشن هایی که داخل سامانه وجود دارد هدایت می کند.

برای مثال فرض کنید که می خواهیم روی دامنه articlesland.ir وب اپلیکیشن ها را مستقر کنیم. باید وب سرور را طوری تنظیم کنیم تا وقتی که به دامنه اصلی درخواست خورد، درخواست را به سمت سرور کنترل کننده اپلیکیشن سمت کاربر ارسال کند، اگر به زیر دامنه server.articlesland.ir درخواست خورد به اپلیکیشن اصلی سمت سرور هدایت کند. البته به یاد داشته باشید که باید داخل کد های React آن آدرسی که بعنوان آدرس سرور اصلی ذخیره شده بود را به server.articlesland.ir تغییر دهیم.

طبق ملزومات ایمنی بیان شده برای قضیه خود استنادی یا self-documenting می توانیم از زبان TypeScript استفاده کنیم که درواقع زبانی است بسیار مشابه به JavaScript اما خیلی از برنامه نویسان آن را زبان مستقلی نمی دانند و این تصورشان از این بابت است که درواقع کد های TypeScript در نهایت به JavaScript کامپایل می شوند. ویژگی این زبان strongly type یا حساس به نوع بودن آن است زیرا زبان JavaScript یک زبان weakly type یا غیر حساس به نوع است و این حساس بودن به نوع باعث شده است که دارای مفاهیم خیلی قوی تری در بحث شیء گرایی داشته باشد. به همین دلیل است که فریم ورک هایی مثل Angular یا Nest به سمت آن تمایل دارند و پایه کد های خود را TypeScript قرار داده اند زیرا الگوی های طراحی آن ها بسیار وابسته به مباحث شیء گرایی است. از طرفی ویژگی خود استنادی بسیار مشهود است زیرا وقتی یک تابع را می خواهید استفاده کنید، IDE برای شما نمایش می دهد که چه نوع متغیری باید بعنوان ورودی بدهید و در خروجی چه نوع و مقداری را موقع داشته باشید.

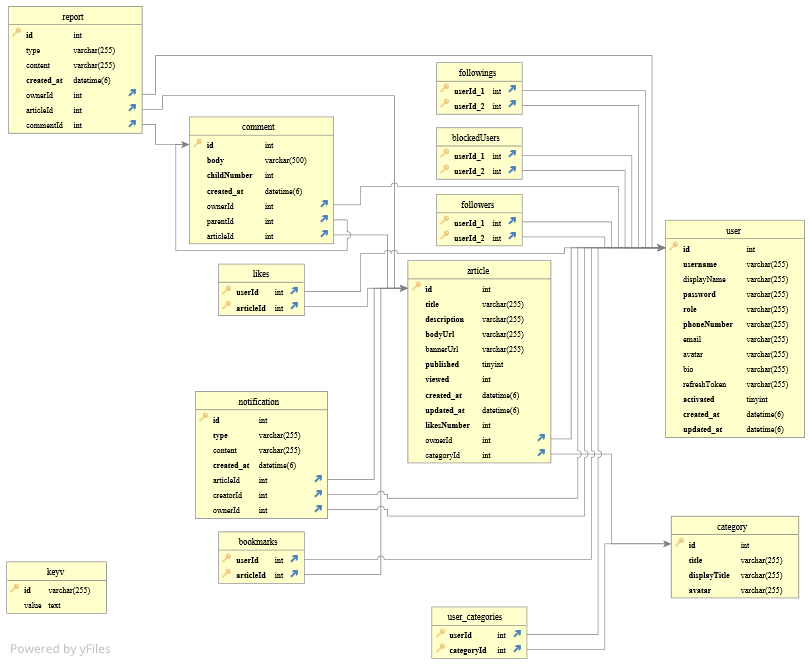
طبق ملزومات ایمنی و امنیتی، برنامه نویسی به سبک TDD و BDD به برنامه نویسان کمک می کند که خودشان را بیشتر به فرهنگ DevOps نزدیک کنند و از ابزار های خودکار سازی استقرار و کشف مشکلات به خوبی بهره ببرند زیرا وقتی برنامه نویس برای کد هایش تست بنویسد به خوبی سرویس های اپلیکیشن را درک می کند و قابلیت اطمینان و تحمل خرابی را افزایش می دهد. از طرفی ابزار های CI/CD نیز می توانند تست های برنامه نویسان را اجرا کنند و در صورت نداشتن مشکل روی branch اصلی merge یا ادغام کنند.

همچنین برنامه نویسان می توانند با اضافه کردن سیستم logging یا ثبت اشکالات احتمالی، قابلیت اطمینان و تحمل خرابی را بسیار بالا ببرند. طبق سیاست های شرکت IBM هیچ اپلیکیشنی نیست که سیستم logging نداشته باشد زیرا log ها یا اشکال های ثبت شده هستند که ما را برای شناسایی خرابی سیستم راهنمایی می کنند تا بتوانیم در اسرع وقت آن ها پیدا کنیم و حلشان کنیم.

طبق ملزومات امنیتی نیز می توانیم به مستندات حملات شایع و معروف که در سامانه های تحت وب رخ می دهند را مطالعه کنیم و برای هر قسمت از آن به دنبال ایجاد راهکارهای امنیتی در چرخه توسعه نرم افزار باشیم. برای مثال من از نرم افزار synk برای بررسی dependency ها یا وابستگی های نرم افزار (ماژول هایی که به کمک node package manager نصب کرده ایم) استفاده کرده ام که سعی کنم همیشه ورژن ماژول های استفاده شده را به اخرین نسخه تبدیل کنم تا موجب نقص امنیتی از نوع Vulnerable and Outdated Components نشود. همچنین سیستم logging بیان شده می تواند از کمبود اطلاعات برای رفع مشکلات اساسی سیستم جلوگیری کند که این نوع نقص امنیتی را با نام Security Logging and and Monitoring Failures می شناسند.

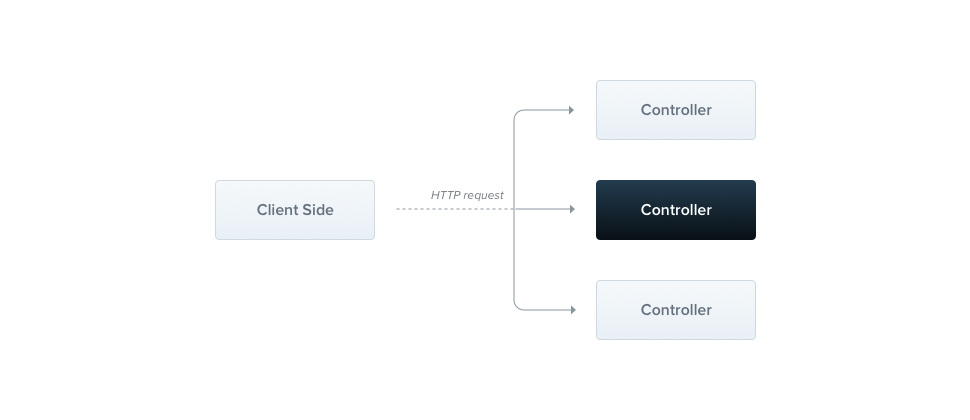
پس از بررسی الزامات SRS و انتخاب های درست و هوشمندانه برای پیاده سازی و توسعه کد های نرم افزار، نوبت بررسی روال اصلی پیاده سازی است.

روال کار من که یک برنامه نویس Full Stack هستم بر این اساس است که ابتدا طرح ظاهری اپلیکیشن سمت کاربر را با یک طراح رابط کاربری و تجربه کاربری حرفه ای ، آماده می کنم. که ما از نرم افزار تحت وب Figma استفاده کردیم.

سپس از روی رابط کاربری طراحی شده، سعی در کشیدن نمودار Entity Relationship Diagram می کنم تا بتوانم دقیق تر برای سرویس ها و داده های دریافتی از سمت کاربر در اپلیکیشن سرور تصمیمی گیری کنم. تصویر نمودار ERD نهایی را در تصویر زیر مشاهده می کنید.

پس از طراحی ERD ، یک پروژه Nest.js ایجاد می کنم و دیتابیس را با ORM یا Object-relational Mapping پیاده سازی میکنم. ORM درواقع یک کتابخانه است که می توانیم به کمک آن بدون اینکه مستقیما table ایجاد کنیم یا حتی روی table مورد نظر به شکل مستقیم query بزنیم، از توابع مربوط به آن برای جایگزین استفاده کنیم. برای ایجاد انتزاع می توانیم یک table را class و هر یک از ردیف یا record های table را، یک object یا شیء ببینیم.

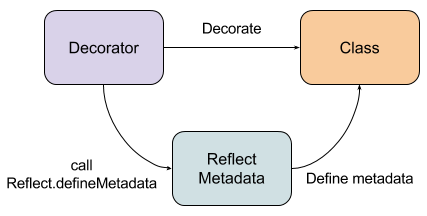
سپس نوبت پیاده سازی API ها است. در nest.js مفهومی وجود دارد به اسم controller که این کنترلرها مسئول رسیدگی به درخواست های دریافتی و بازگرداندن پاسخ ها به مشتری هستند.



هدف یک کنترلر دریافت درخواست های برنامه است. مکانیسم مسیریابی را کنترل می کند به طوریکه مشخص می کند چه کنترل کننده ای چه درخواستی را دریافت می کند. اغلب، هر کنترل کننده بیش از یک مسیر دارد و مسیرهای مختلف می توانند اقدامات مختلفی را انجام دهند.

برای ایجاد یک کنترلر اولیه، از کلاس ها و دکوراتورها استفاده می کنیم. Decorators کلاس‌ها را با ابرداده‌های مورد نیاز مرتبط می‌کنند و Nest را قادر می‌سازند تا یک نقشه مسیریابی ایجاد کند (درخواست‌ها را به کنترل‌کننده‌های مربوطه گره بزند).

دکوراتورها یک الگوی طراحی و مفهوم شناخته شده در بسیاری از زبان های برنامه نویسی رایج هستند، اما در دنیای جاوا اسکریپت، آنها هنوز نسبتا جدید هستند. درواقع توابعی هستند که در هنگام نمونه گیری از یک کلاس خاص، تغییراتی روی شیء نمونه اعمال می کنند بعد شیء نهایی را بر میگردانند.

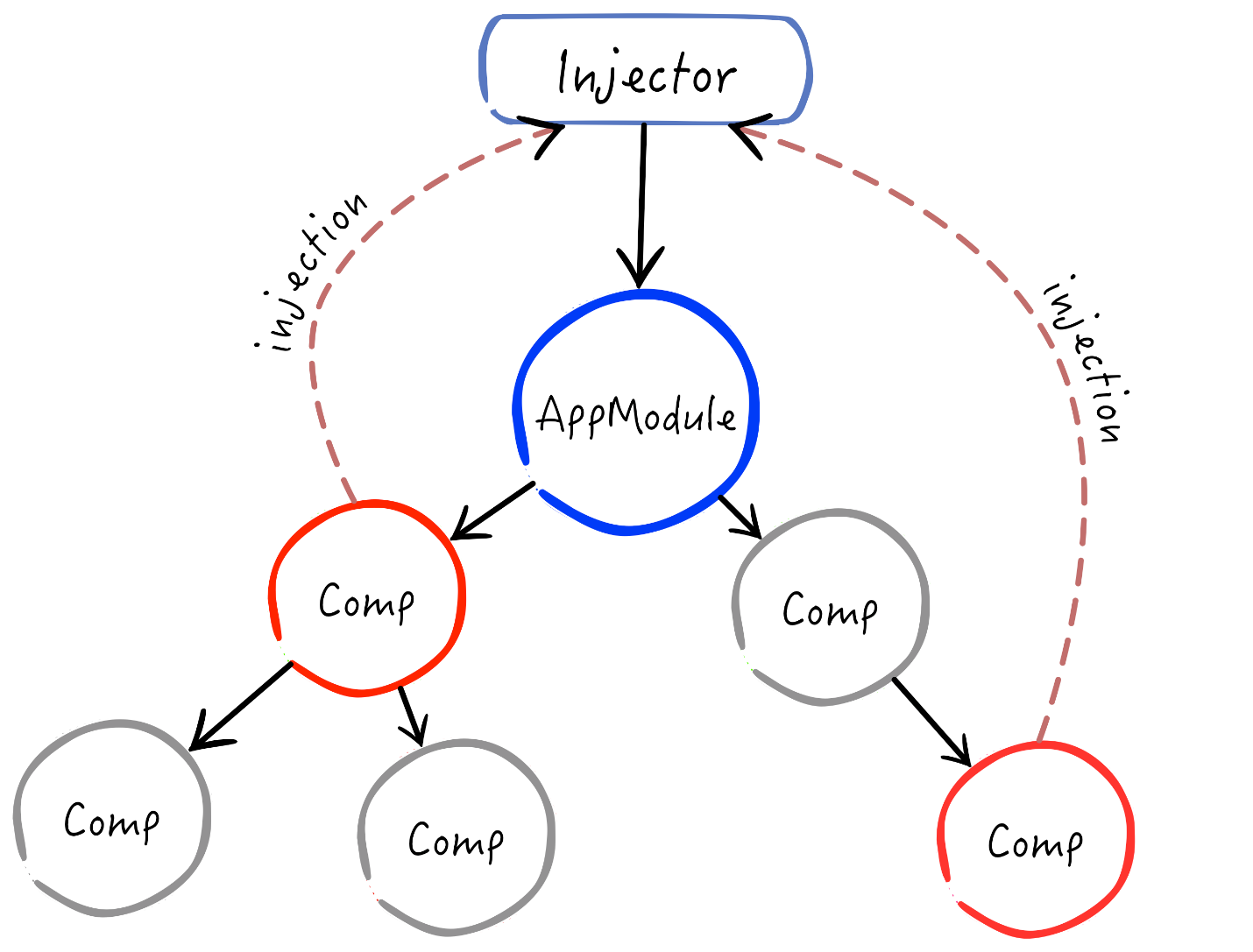


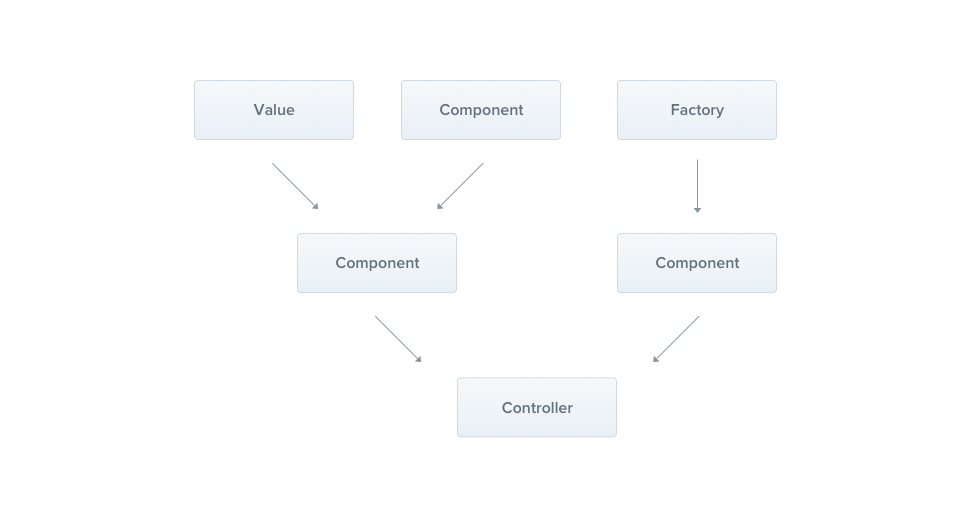
این الگوی طراحی بسیار برای الگوی طراحی singleton مفید است زیرا زمانی که می خواهیم از کل اپلیکیشن سمت سرور یک شیء بگیریم و توسط node.js اجرا کنیم، ویژگی ها و فراداده هایی را به شیء کلاس مورد نظر اضافه خواهد کرد.

سپس این کنترل کننده ها تعدادی تابع را صدا میزنند که این توابع عملکرد های اساسی و پایه سیستم را در بر دارند. نام این توابع ، سرویس یا service است. مجموعه ای از توابع که مربوط به عملکرد های اساسی یک موجودیت است را یک سرویس می نامند که درواقع یک کلاس است. اما سرویس در nest.js اشاره به مفهومی دارد به نام provider یا ارائه دهنده. ارائه دهندگان یک مفهوم اساسی در nest هستند. بسیاری از کلاس‌های nest ممکن است به ‌عنوان یک ارائه‌ دهنده در نظر گرفته شوند – سرویس ها، مخازن، کارخانه‌ها، کمک‌کنندگان و غیره (که جلوتر به این مفاهیم میپردازیم). ایده اصلی یک ارائه دهنده این است که می تواند به عنوان یک وابستگی تزریق یا Inject شود. این واژه بر میگردد به الگوی طراحی که با نام Dependency Injection Design Pattern یا الگوی طراحی تزریق وابستگی ها شناخته می شود.

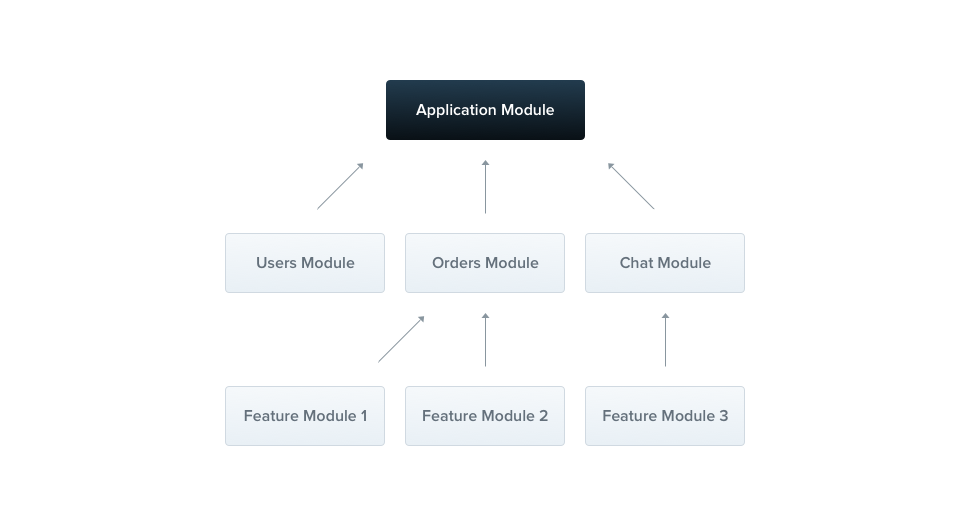
در مهندسی نرم افزار، تزریق وابستگی یک الگوی طراحی است که در آن یک شی یا تابع، اشیاء یا عملکردهای دیگری را دریافت می کند که به آن بستگی دارد.

در اینجا یک کنترل کننده امکان دارد از سرویسی استفاده کند که مربوط به یک کنترل کننده دیگر بوده است پس این الگوی طراحی نقش بسیار مهمی را ایفا می کند.





مفهوم و اجزای مهم بعدی در کد های nest.js، مفهوم ماژول است. این مفهومی بسیار مهم است. برنامه های nest همگی دارای یک ماژول ریشه هستند که در واقع کلاس پایه ای است که از آن یک نمونه گرفته می شود. این ماژول پایه می تواند مجموعه از ماژول ها را در بربگیرد و آن ماژول ها، ماژول های دیگر را، این روند شبیه یک درخت است که یک ریشه دارد و بقیه اجزا به آن متصل می شوند. هر ماژول کنترل کننده یا کنترل کننده ها و سرویس یا سرویس های مخصوص به خود را دارند. اما ماژول ریشه معمولا فقط ماژول های دیگر را داخل خود import یا وارد می کند. ماژول ها اجزای کلیدی هستند، زیرا وظیفه اشتراک گذاری و inject کردن ارائه دهندگان به دیگر کنترل کنندگان سامانه وظیفه این بخش از اجزای nest است.

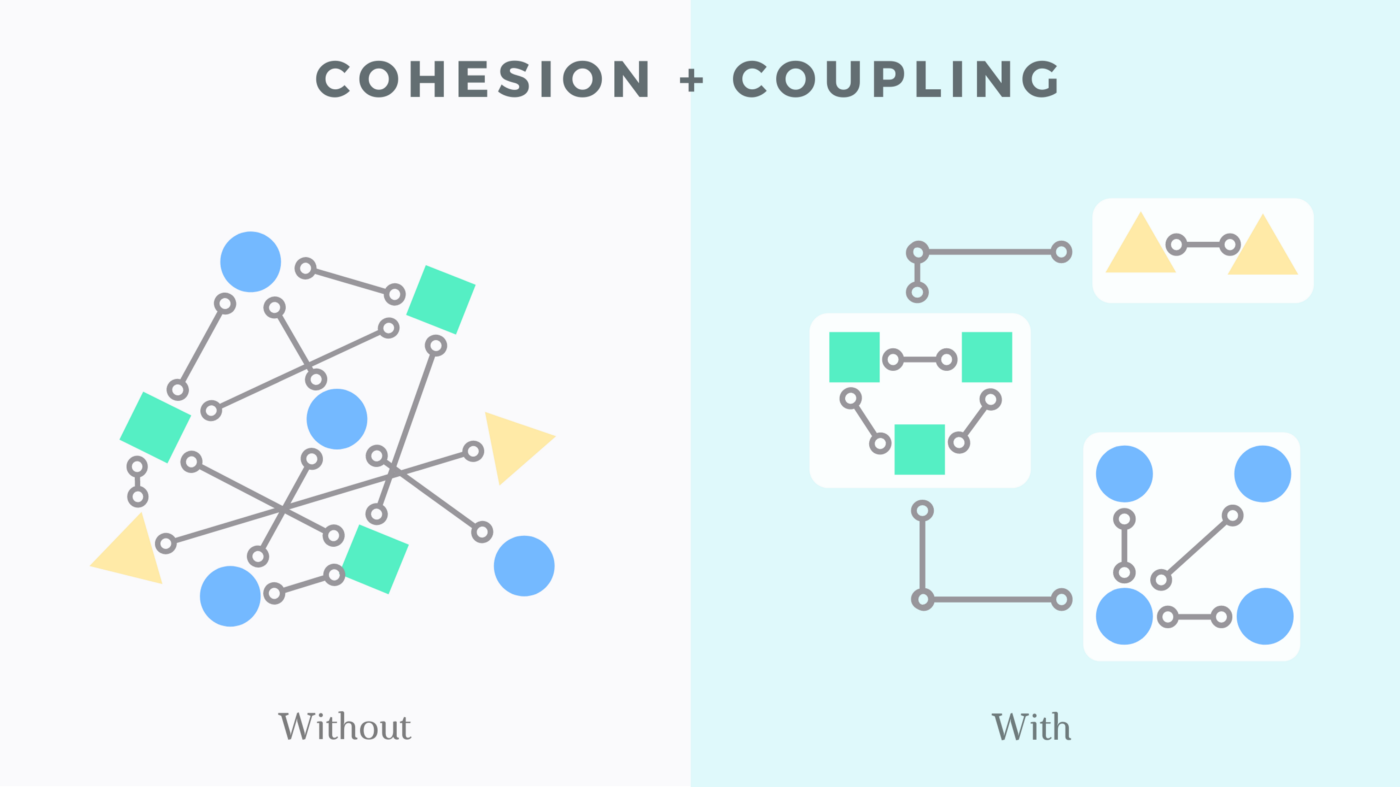


فرض کنید می خواهید برای دو موجودیت مقالات و نظرات REST API طراحی کنید. خب عاقلانه است که مسیر API های مقالات برای عملیات CRUD یک مسیر ثابتی باشد و برای نظرات هم مسیر ثابت دیگری در نظر میگیریم. پس باید دو کنترل کننده مجزا داشته باشیم ، اما مسلما هر دوی آن ها امکان دارد از سرویس های مشترک و غیر مشترکی استفاده کنند. سرویس های مشترک را می توانیم طبق تصویر داخل Application module وارد یا import کنیم که یک نمونه شیء گرفته شود و بین دو کنترل کننده به اشتراک گذاشته شود ولی سرویس های غیر مشترک نباید به یکدیگر مربوط باشند و قابل دسترسی باشند پس یک ماژول جداگانه برای آن ها ایجاد میکنیم و سپس آن دو ماژول را داخل ماژول اصلی import یا وارد می کنیم ولی دیگر نیازی نداریم سرویس ها را داخل Application Module وارد یا import کنیم زیرا می توانیم به وارد کردن مستقیم یکی از ماژول های مقالات یا نظرات داخل دیگری، به سرویس آن ماژول دسترسی داشته باشیم.

اما این دیدگاه ایزوله کردن حداکثری و ارتباط یا درگیری حداقلی کد ها را مفاهیم loose coupling و high cohesion نام دارد.

Loose coupling به این معناست که تعدادی از اجزای نرم افزار طوری در هم تنیده نباشند که اگر بخواهیم یکی از اجزای نرم افزار را تغییر دهیم، مجبور نشویم کل نرم افزار را تغییر دهیم و شاهد به هم خوردن کارکرد اجزای دیگر نرم افزار نباشیم.

High cohesion به این معناست که هر جزئی اگر دارای عملکرد هایی است ، حتما دسترسی به عملکرد های خاص خودش را داشته باشد نه اینکه مجبور باشیم برای استفاده از توابع مخصوص یک موجودیت، سراغ موجودیت دیگر برویم زیرا تابع مورد نظر مربوط به موجودیت دیگر نیست.



اجزا و مفهوم مهم دیگر در nest.js، مفهوم repository است. همانطور که در ابتدای پیاده سازی گفتم، بعد از رسم نمودار های ERD به دنبال پیاده سازی Entity ها به شکل کلاس میروم زیرا منطق ORM به این گونه است. ORM هایی مثل Typeorm یا Mongoose می توانند قابلیت های ویژه ای برای entity های پیاده شده قائل شوند. این ویژگی ها به کمک repository یا مخزن قابل پیاده سازی است.

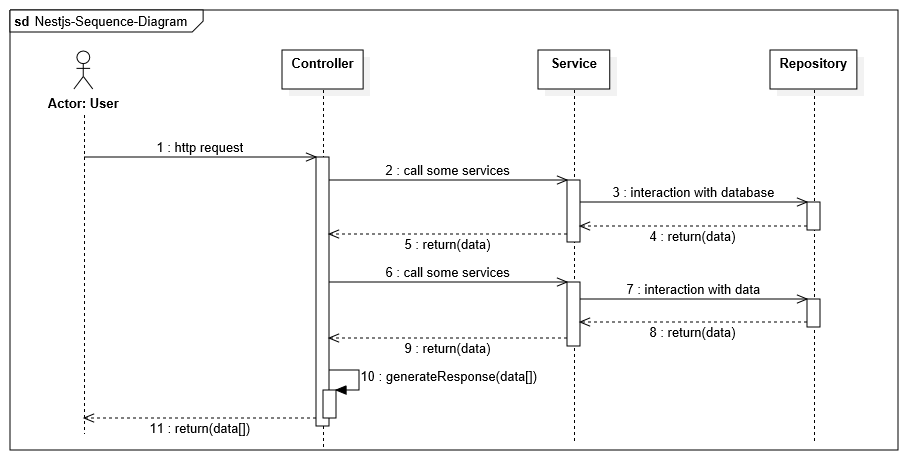
مخزن یا respository که در اینجا نام برده می شود یک الگوی طراحی یا design pattern است. این الگوی طراحی به طوری طراحی شده است که یک کلاس است و وقتی یک entity یا موجودیت پایگاه داده را به آن ورودی می دهیم، بعنوان خروجی یک شیء خواهیم داشت که شامل توابع بسیار مهمی است. این توابع در واقع همان query یا پرس و جو های پایگاه داده است، برای مثال تابع find که برای سطر های داخل جدول یا table ها است یا تابع create که برای ایجاد یک سطر جدید در جدول یا table ها است.



ریپازیتوری ها مستقیما داخل سرویس ها inject یا تزریق می شوند تا مستقیم از طریق سرویس ها قابل استفاده باشند.

همانطور که بیان شد روال ارسال تا دریافت پاسخ از سمت اپلیکیشن سمت سرور به شرح زیر است:

ابتدا درخواست به اپلیکیشن سمت سرور میرسد، سپس کنترل کننده مربوط به آن مسیر درخواست زده شده، اجرا می شود، هر کنترل کننده، سرویس دهنگان مربوط به عملیات خود را به ترتیب اجرا می کنند و سپس داده های خروجی را آماده و برای اپلیکیشن سمت کاربر ارسال می کند تا نمایش داده شود.

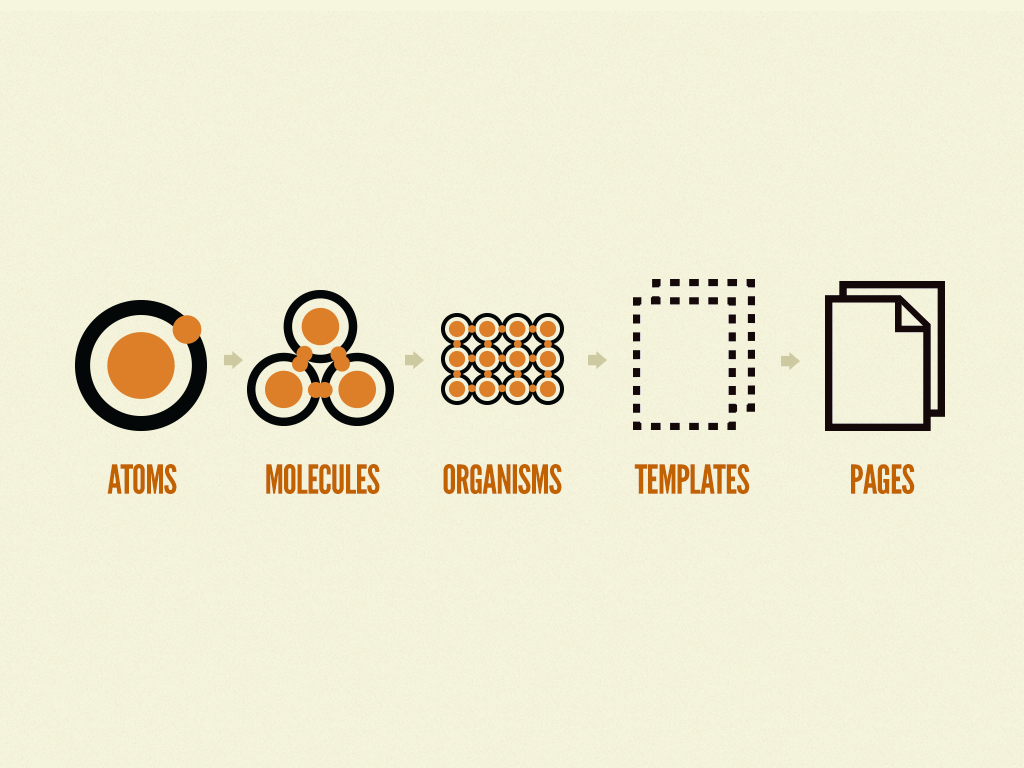


در ادامه به توسعه API هایی می پردازیم که در sprint پیش رو مورد نیاز است و به شکل تسک پیاده شده است. این روالی که توضیح داده شد تا مراحل انتهای روال توسعه نرم افزار ادامه خواهد داشت.

یادمان باشد که باید API های پیاده شده را به کمک Swagger مستند کنیم تا برنامه نویس سمت کاربر بتواند به راحتی با API ها کار کند و پس از درک بررسی عملکرد API ها ، آنها را به اپلیکیشن سمت کاربر متصل کند.

توضیح اپلیکیشن سمت کاربر: ابتدا باید UI طراحی شده را به شکل کد پیاده سازی کنیم. تبدیل طرح به کد مستلزم رعایت کردن یک سری استاندارد است ولی عموما سلیقه ای به آن عمل می کنند و به عنوان یک senior frontend developer بسیار مشاهده کرده ام که حتی یک فرد با ۷ سال سابقه کار هنوز هم ملاک اصلی خود را مفاهیم ساده یعنی کامپوننت قرار میدهد ولی کاملا سلیقه ای هر اجزای صفحه را کامپوننت میبیند.

شخصا از Atomic Design Pattern استفاده می کنم که یک الگوی طراحی یا Design Pattern است:



کد های فرانت را می توانیم به شکل بالا تجزیه و تحلیل کنیم: (اسامی فنی داخل پرانتز آمده است)

* اتم ( یا component ) یعنی کوچک ترین ساختار های یک وب اپلیکیشن برای مثال یک دکمه در صفحه که آن دکمه در استاندارد طراحی UI باید چند نوع مشخص شده باید و در صفحات مختلف با رنگ های مختلف استفاده می شوند.
* قسمت بعدی مولکول ( یا Higher Order Component ) است که می توانیم به یک کامپوننتی اشاره کنیم که خود حاوی چند کامپوننت کوچک است منتها در چندین صفحه دیگر امکان دارد استفاده شود.
* ارگانیسم ( یا Container ) یک کامپوننت های ریز و درشت در کنار هم قرار گرفته شده است و در حدی کوچک نیست که بتوان گفت در چندین صفحه استفاده می شود و لزوما اتفاقا نباید بشود زیرا عمدتا داده ها در سطح container ها هستند که از سمت اپلیکیشن بکند گرفته می شود و به کامپوننت های زیر دست داده می شود. دقیقا خود کلمه ارگانیسم هم به وضوح همین نیت را گواهی میدهد.
* قالب ( Layout ) در بر گیرنده چندین حالت از container ها است. به شکلی که یک سری container ها ایستا هستند و تغییر نمیکنند و یک سری دیگر پویا. برای مثال یک صفحه داشبورد را تصور کنیم که یک navbar در بالای صفحه. یک sidebar در سمت چپ یا راست وجود دارد و داخل آنها مجموعه ای لیست ها ، دکمه ها و کامپوننت های دیگر. تا وقتی شما در صفحه داشبورد هستید یک قسمت از صفحه است که با کلیک بر روی دکمه ها محتوای آن تغییر می کنند.
* صفحه ها نیز شامل شاکله و بدنه اصلی صفحه هستند. هرچیزی که در یک صفحه مشاهده می کنید به ترتیب اولویت از بالا به پایین به شکل container قرار میگیرند. هر صفحه لزوما یک قالب ممکن است نداشته باشد همانطور که یک ارگانیسم میتواند حتما یک مولکول نداشته باشد و مستقیما یک اتم درون خود جای دهد.

هر کدام از اجزایی که توضیح دادیم از یک قاعده کلی در ساختار کد پیروی می کند.

در ابتدای فایل کد که فایل ها و کامپوننت های import یا وارد شده اند.

سپس یک تابعی وجود دارد که نمایانگر کامپوننت اصلی داخل این فایل کد است.

داخل این تابع یک return یا بازگشت وجود دارد که اگر مقدار بازگشتی آن را چک کنیم ، یکسری المنت های HTML مشاهده می کنیم. به این شکل از مقادیر که توسط تابع react برمیگردد، JSX گفته می شود. JSX یک قائده نوشتاری المنت های HTML است که داخل JavaScript نوشته می شود.

تعدادی توابع دیگر و مفاهیم دیگر نیز وجود دارد که در بدنه تابع و قبل از return یا همان بازگشت، نوشته می شوند. برای مثال state ها معادل متغیر ها هستند با این تفاوت که مستقیما در JSX استفاده می شوند و مقدار آن ها داخل صفحه نمایش داده می شود. اگر یک state مقدارش تغییر کند، کل آن کامپوننت مجددا render می شود و برای کاربر مقدار جدید را نمایش می دهد.

همچنین یک مفهومی وجود دارد به اسم hook که مجموعه یکسری توابع است که در سطح کامپوننت استفاده می شود. هرکدام از hook ها عملیات خاصی را انجام میدهد برای مثال هوک useState یک state ایجاد می کند همراه با تابعی که آن متغیر را مجددا مقدار دهی میکند.

یا یک هوک کاربردی دیگر مثل useEffect که پس از بارگذاری کل صفحه، یکبار اجرا خواهد شد و هر تابعی که بعنوان ورودی به آن داده باشید را اجرا می کند. البته این هوک میتواند یکسری ورودی بعنوان لیست dependency ها قبول کند که در این صورت منتظر می ماند تا روی dependency های مورد نظر اتفاق جدیدی روی دهد. برای مثال اگر یک متغیر را بعنوان ورودی به هوک بدهید، منتظر میماند تا آن مقدار تغییر کند تا تابع مشخص شده یکبار دیگر اجرا شود.

این شرح بسیار کوچک و ساده از ساختار کد های React بود که در مراحل طراحی کامپوننت و اجزای دیگر مدام تکرار می شود.

نکته: هر اجزای بزرگ تر در بر گیرنده اجزای کوچک تر است پس اجزای بزرگتر نسبت به اجزای کوچک تر رابطه پدر فرزندی دارند پس اجزای بزرگتر می توانند state های خود را با اجزای کوچکتر به اشتراک بگذارند. برای مثال می توانیم state یک container را داخل یک component استفاده کنیم.

امروزه می توانیم از UI framework ها برای پیاده سازی هرچه راحت تر طرح های رابط کاربری و پیاده سازی های رابط کاربری استفاده کنیم. UI framework های متعددی برای React موجود است. مثل:

* Material UI
* Antd
* Chakra
* Mantine
* Boostrap
* Tailwind

و دیگر فریم ورک های رابط کاربری.

در این پروژه از Mantine استفاده شده است که مستندات آن هم برای Figma و هم برای React موجود است. درواقع فریم ورک های رابط کاربری دارای یکسری کامپوننت بسیار کاربردی و از پیش تعریف شده هستند که برنامه نویس می تواند به راحتی از آن استفاده کند و حتی برای استفاده خاص تر آن را وابسته به طراحی رابط کاربری خود ، ویرایش کند.

پس به طراحی رابط کاربری می پردازیم و پس از تکمیل کردن تسک های تعریف شده در sprint پیش رو، نوبت به اتصال اپلیکیشن سمت کاربر به API های پیاده سازی شده است.

برای ارسال درخواست ها می توانیم از کتابخانه ای به اسم Axios و البته مستندات APIS ها استفاده میکنیم.

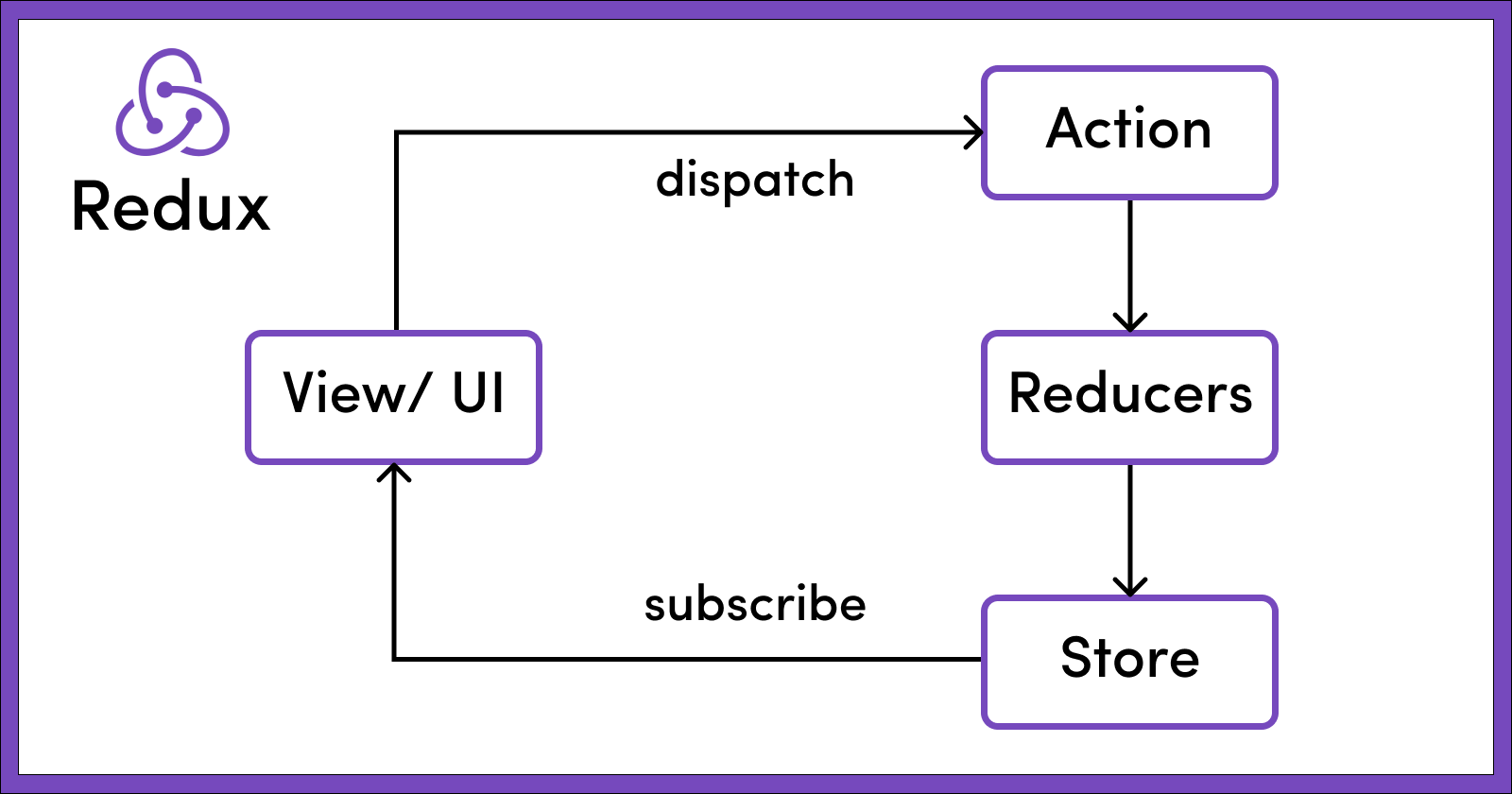
برای ارسال درخواست و استفاده از پاسخ برای render کردن مجدد کامپوننت ها ، باید یک سطح از کامپوننت ها بالاتر برویم. یک سطح بالاتر می شود یک container که در بر گیرنده تعدادی از کامپوننت ها می باشد. بنده عموما بر حسب تجربه، توابعی که درخواست ها را ارسال می کنند و منتظر پاسخ می مانند را در سطح container قرار می دهم که در صورت بروز رویداد یا حتی بارگزاری شدن کامل صفحه ، آن دسته از کامپوننت هایی که نیاز به CSR دارند ، container آنها به اپلیکیشن سمت سرور درخواست بزند و در صورت بازگشت پاسخ از سمت سرور، آن داده ها را داخل کامپوننت ها قرار دهد و مجدد آن ها را نمایش دهد. برای رویداد یا event ها نیز این موضوع برقرار است که میتوانیم روی یک دکمه منتظر عمل کلیک شدن باشیم تا عملیاتی را انجام دهیم ولی باید یادمان باشد که حتما در سطح container باید رخ دهد.

هرچند گاهی ممکن است که اطلاعاتی که از سمت سرور دریافت می کنیم و داخل یک state ذخیره می کنیم را بخواهیم داخل یک container دیگر استفاده کنیم که با container فعلی هیچ رابطه ای ندارد، در این شرایط باید از روش های دیگری مثل state management استفاده کنیم.

ما کتابخانه های متفاوتی برای پیاده سازی state management داریم مثل:

* Redux
* Recoil
* MobX
* useContext
* و دیگر موارد

اما یکی از رایج ترین و محبوب ترین کتابخانه ها Redux است. این کتابخانه از الگوی طراحی Reactor برای ذخیره و واکشی داده ها استفاده می کند که در ادامه شکل مربوط به آن را مشاهده خواهید کرد.



این روش به برنامه نویس کمک می کند که state را بین چندین component که ارتباط والد نسبت به یکدیگر ندارند ، به اشتراک بگذارد. برای مثال عملی می توانیم به input قسمت جستجو داخل navbar اشاره کنیم و container دیگر در صفحه جستجو که مسئولیت درخواست و نمایش مقالاتی را دارد که با آن کلیدواژه جستجو شده است. این کلیدواژه باید بین دو container که ارتباطی به یکدیگر ندارند اشتراک گذاشته شود.

تا این قسمت اجزای مختلف کد های سمت کاربر را توضیح مختصری دادیم و نحوه پیاده سازی آن را مرور بسیار کوتاهی کردیم. زمانی که اتصالات با اپلیکیشن سمت سرور و نمایش اطلاعات در صفحه به خوبی انجام شد. می توانیم تست نهایی را انجام دهیم و پس از اتمام sprint، به sprint بعدی برویم.

این بود شرح مختصری از نحوه پیاده سازی پروژه توسط بنده.