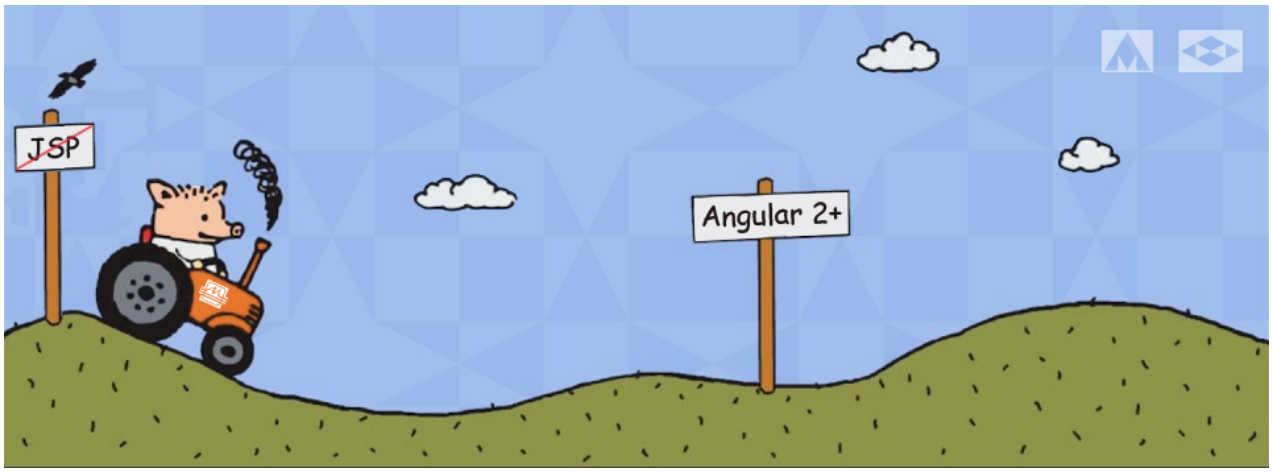
Семилетними шагами: миграция с JSP + Angular JS на Angular 2+

- Блог компании Миландр,
- Разработка веб-сайтов,
- JavaScript,
- Angular,
- TypeScript



Что нужно для перехода от серверного рендеринга к пользовательскому? Чем хорош **Angular 2+** и как на него перейти? В этой статье попытаемся разобраться в данных вопросах и описать процесс миграции от серверных технологий рендеринга, таких, как **JSP**, к клиентским технологиям рендеринга представлений с использованием **Angular**.

Используемые технологии

Прежде чем приступить к коду, опишем весь стек используемых технологий:

- **Spring Framework** - фреймворк для **Java**-платформы.
- **JSP** - технология, позволяющая создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница **JSP** содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов (например, **HTML**) и **JSP**-элементы. Эти элементы конструируют динамическое содержимое, позволяя внедрять **Java**-код в статичное содержимое страницы.
- **AngularJS** и **Angular** - **JavaScript**-фреймворки от компании Google для создания клиентских приложений. **AngularJS** был одним из первых **JavaScript**-фреймворков, разработанным для создания одностраничных приложений (SPA). Он был выпущен в 2009 году как фреймворк с открытым исходным кодом. В 2016 году был выпущен **Angular 2**. Он был переписан с нуля на **TypeScript** и не является обратно совместимым с **AngularJS**.

Изначально у нас есть приложение, написанное на **Spring+JSP** с использованием **AngularJS**. Наша цель - перейти к SPA с использованием современной версии **Angular**.

Какие перспективы?

Angular является современным и популярным фреймворком и обладает рядом преимуществ по сравнению с **JSP** и **AngularJS**.
Основное концептуальное различие между **JSP** и **Angular** заключается в том, динамически ли генерируется страница в браузере (на стороне клиента) или на сервере. При использовании **Angular** не приходится извлекать всю страницу с сервера после ее загрузки в первый раз. Вы просто обновляете части страницы, которые изменились на клиенте (хотя это часто происходит в ответ на данные, полученные с сервера). Рендеринг на стороне клиента позволяет избежать ненужных запросов на полную страницу, когда изменилась только ее часть.

Внедряем технологию

Перейдем непосредственно к описанию процесса перехода и рассмотрим вопросы, возникающие при смене технологий.

Инициализация нового Angular приложения

Вместо **JSP**-страниц нам теперь понадобится **Angular** приложение. Для его создания удобно использовать **Angular CLI**. **Angular CLI** - официальный инструмент для инициализации и работы с **Angular**-проектами. Он упрощает создание приложения, его компиляцию.
Angular CLI можно установить, выполнив следующую команду:

```
npm install @angular/cli
```

После создадим новый Angular проект с помощью команды:

```
ng new <название проекта>
```

Команда *ng new* генерирует “скелет” будущего приложения. **Angular CLI** создает одноименную директорию и помещает в нее исходные файлы "скелета" приложения.

Создание служебных сервисов

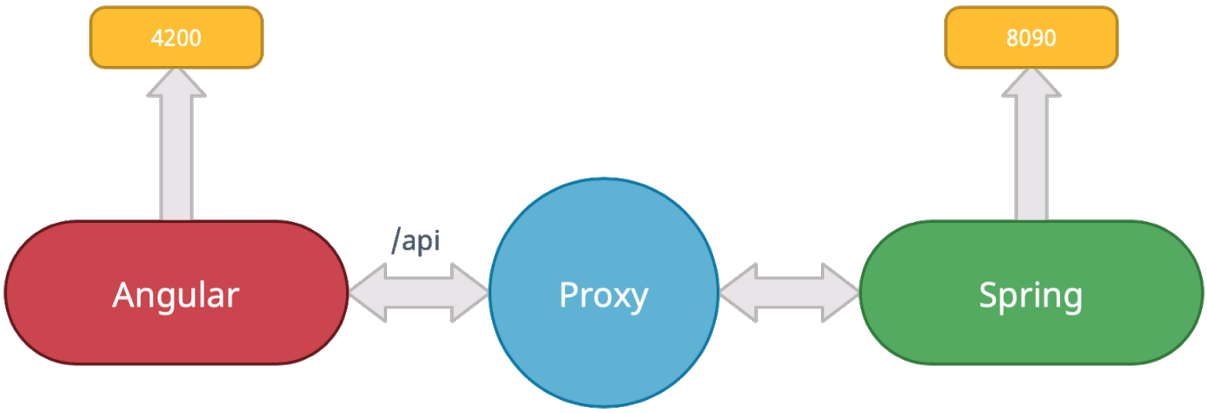
На **JSP**-страницах использовалась информация, содержащая **Java**-выражения, а также специфичные теги: http://www.springframework.org/tags, в частности, для локализации и http://java.sun.com/jsp/jstl/core. Также в некоторых ситуациях может существовать необходимость в кастомизируемом ролевом доступе, информацией о котором, помимо сервера, должен владеть и клиент. Для получения этой информации создадим дополнительные контроллеры с методами, возвращающими нужную информацию. Запросы, возвращающие сообщения для локализации, должны быть доступны без аутентификации. Это связано с тем, что эти сообщения отображаются на всех страницах, включая доступные неавторизованным пользователям. Поправим конфигурацию **Spring Security,** чтобы она игнорировала запросы по шаблону *"/messages/*"*:

```
@Override
public void configure(WebSecurity web) {
    web.ignoring().antMatchers("/messages/*");
}
```

Для того, чтобы использовать эти данные, на фронтенде должны быть заранее известны коды сообщений и названия привилегий. Совпадение информации на фронтенде и бэкенде приходится отслеживать вручную.

Проксирование

В случае, когда клиент хостится на отдельном сервере, возникает потребность в проксировании запросов. Тот факт, что фронтенд и бэкенд прослушивают отдельные порты, не позволяет **Angular** делать внутренние запросы. Это связано с тем, что веб-браузеры разрешают только внутренние запросы к тому же источнику, из которого было загружено веб-приложение. К счастью, мы можем позволить **Angular CLI** выступать в качестве прокси-сервера для нашего бэкенда **Spring**. Приложение **Angular** будет отправлять бэкенд-запросы на сервер разработки, который будет пересылать их на бэкенд.



Для этого добавим в корневую папку проекта файл *'proxy.conf.json'* с конфигурацией прокси-сервера:

```
{
  "/api": {
    "target": "http://localhost:8099",
    "secure": false,
    "cookieDomainRewrite": "localhost",
    "changeOrigin": true
  }
}
```

Также поправим *package.json*, чтобы данная конфигурация применялась, указав в стартовом скрипте "start": "ng serve --proxy-config proxy.conf.json".

Перенос JSP страниц в компоненты Angular

Angular приложение состоит из компонентов, у каждого из которых своя независимая логика. Компонент - полноценная сущность, у которой есть своя логика **TypeScript**, разметка **HTML** и свой стиль **CSS**. Соответственно каждая **JSP**-страница будет разбиваться на отдельные компоненты, в которых в отдельные файлы выносится html, typescript и css. Для отправки запросов к бэкенду создадим отдельные сервисы. Заменим конструкции **AngularJS** на аналогичные в **Angular**.

В каждом компоненте, в методе ngOnInit пропишем получение необходимых сообщений для локализации, а также добавим проверку наличия необходимых для данного компонента привилегий.
При открытии **JSP**-страницы выполняется проверка на наличие привилегий. В случае отсутствия привилегий пользователь перенаправляется на страницу входа.

```
<%
    if (!SecurityUtils.hasRole(SecurityConstants.VIEW_USER)) {
        response.sendRedirect("login.jsp");
    }
%>
```

В Angular есть механизм ограничения перехода, называемый Guards. Guards позволяют ограничить доступ к маршрутам на основе определенного условия.
Все guard-ы должны возвращать либо true, либо false. И происходить это может как в синхронном режиме (тип Boolean), так и в асинхронном режиме (Observable<boolean> или Promise<boolean>). Данный пример разрешает переход при наличии у пользователя привилегии VIEW_SETTINGS.
Метод *hasPrivilege* отправляет запрос на сервис и возвращает Observable<boolean>.

```
@Injectable({
 providedIn: 'root'
})
export class SettingsGuard {
  readonly  SECURITY_CONSTANTS = SecurityConstants;
  constructor(private router: Router,
              private securityConstantService:
  }
```

```
  canActivate(next: ActivatedRouteSnapshot, state:
  Observable<boolean|UrlTree> | Promise<boolean|UrlTree> |
    return
this.securityConstantService.hasPrivilege(this.SECURITY_CONST
ANTS.VIEW_SETTINGS);
  }
}
```

Обработка ошибок

Следующий шаг - обработка ошибок, возвращаемых сервером.
Добавим HttpInterceptor, который обеспечивает перехват http-запросов и ответов для преобразования или обработки их перед передачей. При получении ошибок 401 или 403 будем перенаправлять пользователя на страницу входа:

```
public handleError(err: HttpErrorResponse): Observable<any> {
  if (err.url != null && (err.status === 401 || err.status
=== 403)) {
    this.router.navigate(['/login']);
  }
  return throwError(err.error);
}
```

Подобным образом можно обрабатывать другие ошибки, подробнее можно почитать, например, здесь.

Кэширование

Сообщения для локализации интерфейса, а также ряд других констант не меняется во время использования приложения, поэтому нет смысла каждый раз запрашивать их с сервера. Для реализации кэширования добавим *RequestCache* и *CachingInterceptor*.

RequestCache - сервис, который содержит методы, позволяющие добавлять информацию в кэш и извлекать ее из него. Вариантом реализации может послужить хранение данных в виде пар "ключ/значение", где ключом является название сообщения.

```
cache = new Map<string, RequestCacheEntry>();
```

```
get(req: HttpRequest<any>): HttpResponse<any> | undefined {
    const url = req.url + req.body.key;
    const cached = this.cache.get(url);

    if (!cached) {
      return undefined;
    }

    const isExpired = cached.lastRead < (Date.now() - maxAge);
    return isExpired ? undefined : cached.response;
  }
```

```
put(req: HttpRequest<any>, response: HttpResponse<any>): void
{
    const url = req.url + req.body.key;
    const newEntry = { url, response, lastRead: Date.now() };
```

```
    this.cache.set(url, newEntry);

    const expired = Date.now() - maxAge;
    this.cache.forEach(entry => {
      if (entry.lastRead < expired) {
        this.cache.delete(entry.url);
      }
    });
  }
```

CachingInterceptor реализует интерфейс HttpInterceptor, перехватывает запросы и решает, нужно их кэшировать или нет. Любой класс, реализующий интерфейс HttpInterceptor, должен иметь метод intercept.
В этом методе мы сначала проверим, нужно ли кэшировать данные из запроса (метод *isCacheable*), и, если нужно, попробуем извлечь данные из нашего кэша. Если у нас есть кэшированные данные для конкретного запроса, то мы вернем Observable с этими данными. Иначе мы отправим запрос на сервер для извлечения данных, в нашем случае, используя метод sendRequest. Этот метод также кэширует запрос для дальнейшего использования.

```
intercept(req: HttpRequest<any>, next: HttpHandler) {
  if (!isCacheable(req)) { return next.handle(req); }

  const cachedResponse = this.cache.get(req);
  if (req.headers.get('x-refresh')) {
    const results$ = sendRequest(req, next, this.cache);
    return cachedResponse ?
      results$.pipe( startWith(cachedResponse) ) :
      results$;
  }
```

```
    return cachedResponse ?
      of(cachedResponse) : sendRequest(req, next, this.cache);
}
```

Заключение

Итак, описанные действия приблизили нас к получению современного Angular приложения, совместимого с нашим бэкендом. Остальное было делом техники. Популярные фреймворки предоставляют широкий спектр возможностей и огромное комьюнити с поддержкой. Уход от устаревших технологий необходим для полноценного развития приложения, даже если переход может вызвать сложности. Надеюсь, эта статья поможет вам продвинуться в нужном для вас направлении.

Теги:

- Angular
- AngularJs
- переход на angular
- spring
- веб-разработа
- typescript
- jsp

Хабы:

- Блог компании Миландр
- Разработка веб-сайтов
- JavaScript
- Angular
- TypeScript