# Roteiro de exercícios sobre SaaS, parte de serviços, disciplina Engenharia de Software e Sistemas

## Paulo Borba

## Centro de Informática

## Universidade Federal de Pernambuco

Adicione as respostas imediatamente após as perguntas, usando uma outra cor. Leia com calma e atenção as instruções, e reflita com cautela sobre as consequências de cada comando. **As instruções abaixo assumem que toda a parte de instalação do roteiro de SaaS-GUI foi concluída.** É também assumido que todas as referência (manuais, tutoriais, etc.) mencionadas no roteiro anterior já foram consultadas, e que os alunos devem estar prontos para rapidamente consultá-las novamente durante o resto do curso, inclusive quando seguindo o roteiro abaixo. Os *commits* mencionados abaixo não são referentes ao que está no *master* do repositório do projeto; eles são referentes ao que está no ***branch* criado pelo professor e associado à essa edição da disciplina (por exemplo, 2019-2, 2020-1, etc.)**

## Aula 4

1. No projeto [**teachingassistant**](https://github.com/pauloborba/teachingassistant/commits/master), analise o *commit* "[versao toy do servidor](https://github.com/pauloborba/teachingassistant/commit/bd62070b3f1c37496b6f66a856fde9b1393b5697)". Um dos arquivos no *commit*, o**package.json**, é responsável pela configuração do projeto do servidor. Qual o efeito das declarações que envolvem **"devDependencies" e "dependencies"** nesse arquivo? O arquivo **tsconfig.json** é responsável pela configuração do compilador Typescript. Qual o efeito da declaração **"outDir":"compiledCode"** neste segundo arquivo?
2. O arquivo **ta-server.ts** é um servidor [Express](http://expressjs.com/en/starter/installing.html) simples, escrito em Typescript. Aplique as mudanças desse *commit* no seu *branch* (idealmente com *cherry-pick*). Instale as dependências necessárias digitando **npm install** no diretório **ta-server**. Em seguida, coloque o servidor no ar digitando **npm start**. Qual o efeito da declaração **"start":"npm run tsc && node compiledCode/ta-server.js"** no arquivo **package.json**? Em algumas máquinas pode aparecer o erro **"UNMET PEER DEPENDENCY xxxxx"** quando executando **npm install**. Nesse caso, dê **npm install xxxxx** para cada um dos "xxxxx" que aparecerem e, em seguida, dê **npm install**. Se encontrar problemas de permissão quando tentando instalar algo do node, consulte o [manual](https://docs.npmjs.com/getting-started/fixing-npm-permissions) do node. No Windows, o tsc é instalado no diretório node\_modules/.bin. Você precisa adicionar o caminho desse diretório ao seu PATH. Se necessário, use **^C** para matar um processo que esteja executando em um terminal.
3. Teste o servidor digitando no sew *browser*

**http://localhost:3000/**

Qual o comportamento observado no *browser*? Cole a imagem logo a seguir. Que requisições/serviços (explicitar método HTTP, URL, parâmetros, retorno esperado) esse servidor está apto a receber? O que representa o parâmetro **res** no código do servidor? O que representa o parâmetro **req** no código do servidor? Consulte o manual de referência de [Express](http://expressjs.com) e [Node.js](https://nodejs.org/en/).

1. Verifique como depurar o código do servidor com uma ferramenta de depuração (como a [provida](https://code.visualstudio.com/docs/nodejs/nodejs-debugging) pelo VSCode, por exemplo) ou, em última hipótese, com **console.log().**
2. Analise o *commit* "versao inicial do servidor, apos extracao de aluno para area comum". Observe que a classe **Aluno** foi movida de [**ta-gui/src/app/aluno.ts**](https://github.com/pauloborba/teachingassistant/commits/2019-2#diff-0f57754a05119ef068edaa10de6bec8e) para[**common/aluno.ts**](https://github.com/pauloborba/teachingassistant/commits/2019-2#diff-65f657d20934dfb9ec374ab261888805) já que ela vai ser usada pelo servidor também. Assim, veremos que os projetos **ta-gui** e **ta-server** dependem do projeto **common** (os arquivos de configuração do mesmo fazem parte do *commit*). Sem a mudança de diretório, **ta-server** teria que depender de **ta-gui** para usar a classe **Aluno**. Por que isso poderia não ser uma boa opção de *design*? Além da mudança de diretório, o método **clone** foi modificado. Você acha que essas modificações alteraram o comportamento dos objetos dessa classe (no sentido de que o método **clone** retornava um aluno com características diferentes da que ele agora retorna)? Por que?
3. Observe a semelhança da classe **CadastroDeAlunos** do servidor com a classe [**AlunoService**](https://github.com/pauloborba/teachingassistant/blob/41c886377d61e2db7b89f57ae01edf101146988b/gui/ta-gui/src/app/aluno.service.ts) da GUI. Esquecendo os detalhes de importação e anotação antes das declarações das classes, quais as principais diferenças entre elas? O que justifica essas diferenças? As duas não permanecerão na versão final do sistema. Essa duplicação ocorre apenas porque essa funcionalidade de armazenamento de alunos, que pertence naturalmente ao servidor, estava sendo temporariamente simulada na GUI, para que a gente pudesse testar a GUI de forma isolada do servidor.
4. Observe o arquivo [**ta-server.ts**](https://github.com/pauloborba/teachingassistant/commits/5c594afd0337f11e2329c554151c950767e35e10#diff-e8e3855437d5cb12c83fad60c8473eb7), que implementa o servidor. Essa é uma versão mais elaborada do servidor. Que requisições/serviços (explicitar método HTTP, URL, parâmetros, retorno esperado) esse servidor está apto a receber? O que justifica a diferença entre os métodos escolhidos? Observe que o pacote [body-parser](https://www.npmjs.com/package/body-parser) foi adicionado como dependência do projeto **ta-server** e é utilizado pelo novo código do servidor; consulte a sua documentação. Que funcionalidade é oferecida por esse pacote e utilizada pelo servidor?
5. Note que esse servidor repassa a execução dos serviços para o cadastro de alunos, e fica responsável apenas por extrair informações das requisições, e preparar as respostas no formato adequado. Seria melhor uma solução que não empregasse essa separação entre as preocupações de realizar o serviço (classe **CadastroDeAlunos**), e a de processar requisições e respostas ([**ta-server.ts**](https://github.com/pauloborba/teachingassistant/commits/5c594afd0337f11e2329c554151c950767e35e10#diff-e8e3855437d5cb12c83fad60c8473eb7))? Por que?
6. Aplique as mudanças desse *commit* no seu *branch* (idealmente com *cherry-pick*). Instale as dependências necessárias digitando **npm install** nos diretórios **ta-server** e **common**. Em seguida, coloque o servidor no ar digitando **npm start**. Se você estiver usando o Windows, tem que ajustar (no arquivo **package.json**) o caminho do **tsc**; ele tenta chamar **./node\_modules/.**, mas, no Windows, tem que descobrir o endereço onde o **tsc** está instalado e ajustar no **package.json** (invertendo as barras do PATH, já que essa é uma diferença importante do unix para o Windows). Teste o servidor. As requisições GET podem ser testadas pelo browser, como antes. As requisições POST e PUT podem ser testadas via linha de comandos da seguinte forma:

**curl -i -X POST -H 'Content-Type: application/json' -d '{"nome": "Mariana", "cpf":"683", "email":"mcb@cin", "metas":{}}' localhost:3000/aluno/**

**curl -i -X PUT -H 'Content-Type: application/json' -d '{"nome": "Pedro", "cpf":"684", "email":"pcb@ufpe", "metas":{"requisitos":"MA"}}' localhost:3000/aluno/**

Cadastre e modifique alguns alunos com esses comandos (cuidado com o copy-and-paste direto daqui para o terminal, o que pode introduzir caracteres e quebras de linhas indesejadas; se for o caso, copie antes para um editor na sua máquina, elimine as quebras, etc.). Verifique via *browser* se as operações estão sendo corretamente executadas. Cole logo a seguir a imagem do seu *browser* após a solicitação do serviço GET **/alunos**.Se você estiver usando o Windows (sem cygwin ou WSL, o que certamente atrapalha o andamento do curso), é provável que você tenha que instalar o comando **curl** ou usar um outro software (veja o API Client do [Postman](https://www.postman.com), do [Insomnia](https://insomnia.rest) ou similares) que permita esse tipo de teste através do envio de requisições.

## Aula 5

1. Analise agora o commit "servidor integrado ao cliente". Note que as modificações no código do servidor foram apenas para melhorar a legibilidade, e permitir que a gente teste a parte de GUI e a parte de cliente tanto de forma isolada quanto integrada. Aplique as mudanças desse *commit* no seu *branch* (idealmente com *cherry-pick*). Coloque o cliente (em um terminal, com **npm start** no diretório do cliente) e o servidor (em outro terminal ou aba, com **npm start** no diretório do servidor) em execução. Teste o sistema via GUI integrada ao servidor, cadastrando alguns alunos e avaliações de metas. Cole logo a seguir as imagens do seu *browser* mostrando uma das telas da GUI (a de metas, por exemplo) e o retorno dado pelo servidor à requisição [**http://localhost:3000/alunos**](http://localhost:3000/alunos) feita direto no browser*.*
2. As principais modificações necessárias para a integração entre cliente e servidor estão na parte de GUI, na classe **AlunoService**, que agora de fato chama os serviços do servidor, ao invés de simulá-los. Que parte dessa classe **AlunoService** indica que ela realmente acessará o servidor que discutimos antes?
3. Consulte a referência do pacote [RxJS](http://reactivex.io/rxjs/) e o [tutorial](https://angular.io/guide/observables) de Angular sobre o assunto. Estude principalmente o conceito **Observable** (observável), que representa uma potencial lista (ou *stream*) de elementos de um determinado tipo e que podem ser observados e consumidos por quem tiver interesse. Dizemos que representa uma potencial lista porque, de fato, não há garantia de que tal lista venha a existir. Essa indefinição serve para representar, por exemplo, situações como quando um cliente tenta enviar uma requisição a um servidor. Nesses casos, diversos problemas podem ocorrer: o cliente pode estar temporariamente sem acesso à rede, o servidor pode estar fora do ar, etc. Por isso a tentativa de envio da requisição não garante que haverá uma resposta do servidor. O tipo de retorno dessa operação de tentativa de envio deve então ser um **Observable**, indicando que tanto pode haver uma resposta (no caso de requisições HTTP a lista teria apenas um elemento) quanto pode não haver resposta nenhuma, apenas uma sinalização de erro. Mesmo quando houver uma resposta, não há garantias de que ela estará disponível logo após a criação do objeto **Observable**. Isso é o que permite, por exemplo, que a tentativa de envio de uma requisição a um servidor seja implementada de forma assíncrona. Isto é, o cliente rapidamente envia a requisição, recebe um objeto **Observable**, não fica esperando uma resposta do servidor (que pode demorar, ou nem chegar), e segue executando outras operações, evitando o bloqueio da GUI (por exemplo, o usuário pode digitar os conceitos do segundo aluno mesmo que a GUI ainda não tenha recebido a confirmação de que os conceitos do primeiro foram gravados no servidor). Por que o método **getAlunos** da classe **AlunoService** agoraretorna **Observable<Aluno[]>** ao invés de **Aluno[]** (comoera o caso na implementação anterior, que acessava um *array* local ao cliente e retornava seu conteúdo em um novo *array*)? Qual seria o efeito de tirar o **.pipe(retry(2))** do código de **getAlunos**?
4. Em relação ao método **get** chamado em **getAlunos**, quais os parâmetros adicionais recebidos pelos métodos **put** e **post** chamados, respectivamente, nos métodos **atualizar** e **criar**? Por que eles são necessários nesses dois casos e não no caso do **get**? Como esses métodos foram chamados com o tipo **any**, o que podemos assumir sobre os elementos dos objetos **Observable<any>** a serem retornados por eles? Para transformar **Observable<any>** em **Observable<Aluno>**, e assim ter acesso a mais informações nos componentes da GUI quando as requisições forem adequadamente respondidas, usamos a função **map**. O que faz a função recebida como argumento por **map** nos métodos **atualizar** e **criar**? O que verifica a condição **res.success** feita nessas funções? Por que a verificação é feita usado **success** e não um outro atributo?
5. Com essa organização, a classe de serviço é responsável por preparar as requisições dos serviços do servidor (feitas pelas chamadas aos métodos de **HttpClient**), cuidar do reenvio caso necessário (**retry**), e definir como eventuais respostas a essas requisições serão processadas (**map**). Mas a observação e consumo das respostas, ou de notificações de erro, são feitos nos componentes da GUI. Por que você acha que optamos por essa organização? Pense em que tipo de mensagem de erro você poder emitir em um componente e não pode emitir em uma classe de serviço.
6. Na classe **AlunosComponent**, observe o método **ngOnInit**. O método **getAlunos** invocado antes dava garantia de retorno de um *array* de alunos. Agora temos apenas a garantia de que será retornando **Observable<Aluno[]>**, o que não nos garante muita coisa.Para observar e consumir os elementos de um **Observable** ou eventuais notificações de erro, usamos o método **subscribe**.Nesse caso,quantos argumentos ele recebe? Quais os tipos desses argumentos? O primeiro é utilizado para cada *array* que possa eventualmente ser observado. Qual o efeito esperado? O segundo é utilizado para cada notificação de erro. Qual o efeito esperado?
7. Na classe **AlunosComponent**,qual o propósito do **condicional** (**if**) utilizado no método **criarAluno** desse componente? Por que ele é necessário? Adicione o comando **alert("Já executei o criar e o subscribe!");** na última linha do método **criarAluno.** Recarregue (*reload*) a GUIdo sistema e crie um novo aluno. O sistema mostra primeiro o alerta com a mensagem ou o novo aluno na lista de alunos? O que isso indica em termos de comunicação assíncrona e concorrência?
8. Na classe **MetasComponent**,qual o propósito do **condicional** (**if**) utilizado no método **atualizarAluno** desse componente? Por que ele é necessário?
9. Altere o sistema para que seja possível armazenar também conceitos para as metas "Gerência de Projetos" e "Testes". Compile, execute o cliente e servidor, e teste manualmente a nova funcionalidade. Cole logo abaixo a imagem do seu *browser* dando evidência do correto funcionamento da nova funcionalidade.

## Aula 6

1. Altere o sistema para que ele considere também o login GitHub de cada aluno cadastrado, considerando que esse login também não pode ser duplicado, assim como o CPF. Compile, execute o cliente e servidor, e teste manualmente a nova funcionalidade. Cole logo abaixo a imagem do seu *browser* dando evidência do correto funcionamento da nova funcionalidade.
2. Altere o sistema para que seja possível remover um aluno cadastrado (adicione um botão de remoção ao lado de cada aluno listado no componente **Alunos**, ou adicione um botão de remoção ao lado do botão **Adicionar**). Veja no [tutorial](https://angular.io/guide/http#making-a-delete-request) de Angular como solicitar o envio de requisições **DELETE**. Note que para aceitar parâmetros como parte da URL de solicitação de uma requisição ao servidor, temos que registrá-la da seguinte forma no servidor, no caso em que cpf é o nome do parâmetro a ser passado: '/aluno/:cpf'. No código associado a esse serviço, req.params.cpf pode ser usado para obter o valor passado na requisição, assumindo que req representa a requisição sendo tratada (todo objeto que representa uma requisição tem um atributo params, que é um objeto que mapeia nomes de parâmetros nos seus valores associados). Compile, execute o cliente e servidor, e teste manualmente a nova funcionalidade. Cole logo abaixo a imagem do seu *browser* dando evidência do correto funcionamento da nova funcionalidade.