Relatório

1. Camada de apresentação

A biblioteca AngularJS foi utilizada para implementar a camada de apresentação. Essa biblioteca permite que as páginas sejam atualizadas dinamicamente com grande facilidade. Na página de compra dos produtos, o valor total é atualizado dinamicamente toda vez que os dados da compra são alterados. Para implementar esse comportamento, o esforço foi pequeno pois a biblioteca AngularJS encapsula a complexidade dos eventos e listeners JavaScript.

A biblioteca AngularJS também facilita bastante a comunicação através de AJAX. Utilizando os conceitos de controlador, serviço, configuração, filtros e diretivas a organização e reutilização do código é bem eficiente.

Utilizando os recursos de roteamento da biblioteca AngularJS, podemos facilmente implementar o conceito de Single-Page Application. A programação do History para que os botões "Back" e "Forward" funcionem corretamente fica transparente.

A biblioteca Bootstrap também foi utiizada para implementar a camada de apresentação. Essa biblioteca facilita o desenvolvimento de uma interface responsiva e um visual atrativo.

Para implementar a splash screen do processo de checkout, as bibliotecas PleaseWait e SpinKit foram utilizadas.

O build da camada de apresentação foi realizado utilizando a ferramenta Gulp. As bibliotecas utilizadas e já citadas anteriormente foram adicionadas através da ferramenta Bower.

2. Camada de aplicação

A plataforma Node.js foi utilizada para implementar a camada de aplicacação. Essa plataforma é bastante modular e utiliza a linguagem de programação JavaScript. Os módulos desejados podem ser facilmente adicionados no ambiente através da ferramenta NPM e acessados no código através da função "require".

A camada de aplicação é praticamente uma API REST que será "consumida" pela camada de apresentação anteriormente descrita. Com alguns incrementos, ela também poderia ser acessada por aplicativos mobile.

Os módulos Express e BodyParser foram utilizados para construir a interface da camada de aplicação seguindo os princípios do estilo arquitetural REST.

O módulo nativo HTTPS foi utilizado para implementar a comunicação com a API REST da Moip.

O módulo mysql foi adicionado ao ambiente para que a camada de aplicação possa acessar os dados armazenados no MySQL Server

O módulo winston foi utilizado para facilitar a construção do log da aplicação.

O módulo ws foi utilizado para implementar a comunicação através de Websockets com a camada de apresentação. Esses Websockets foram utilizados para informar aos usuários resultado final de um checkout.

3. Testes

Os testes automatizados foram realizados com ajuda dos módulos Mocha, Nock, Rewire e Sinon.

O módulo Mocha oferece um mecanismo simples para que possamos definir e executar a suite de testes.

O módulo Nock permite que as requisições HTTP ou HTTPS sejam interceptadas para que possamos facilmente "mocar" o comportamento de uma API REST que a aplicação deve consumir. No teste, o comportamento da API REST da Moip foi "mocado" com esse módulo.

O módulo Rewire permite que o conteúdo privado de um módulo Node.js seja testado. Esse módulo foi utilizado para testar algumas funções privadas do módulo checkout.

O módulo Sinon permite que os objetos ou funções sejam monitorados durante os testes. Dessa forma, podemos verificar, por exemplo, se determinadas funções foram ou não chamadas. Além disso, ele permite a criação de mocks.

Na página de compra de produtos, acrescentei alguns botões para testes manuais com os cartões de crédito do sandbox da Moip.

4. Implantação

A aplicação foi implantada no cloud do Heroku. O add-on JawsDB MySQL foi adicionado para ter acesso a uma base de dados no MySQL Server do Heroku. O add-on Papertrail foi adicionado para que os dados do log da aplicação possa ser consultado facilmente ou para que alertas para determinados eventos do log sejam criados.

O Heroku, Travis e GitHub foram configurados para implementar o processo de integração contínua e entregra contínua. Dessa forma, quando um push é feito no repositório da aplicação no GitHub, o Travis clona o repositório, prepara o ambiente de acordo com as informações contidas nos arquivos de configuração, executa os testes automatizados e se tudo estiver correto implanta a aplicação no Heroku.

5. Melhorias

5.1. Implementar o carrinho de compras utilizando sessions, webstorage ou banco de dados.

5.2. Implementar autenticação dos usuários

5.3 Melhorar o tratamento de erros da aplicação

5.4 Utilizar o módulo Gulp Watch para tornar dinâmico o processo de build da camada de apresentação

5.5 Gerar arquivos de mapeamento no processo de minify para um evetual processo de depuração do código JavaScript cliente.

5.6 Utilizar um add-on do Heroku que permite executar os Dynos para escalar a aplicação quando necessário, ou seja, de acordo com a demanda.

5.7 Utilizar uma imagem Docker para melhorar o tempo de build no Travis

5.8 Utilizar o Docker para executar o processo de build do Travis localmente também