Laboratório de Desenvolvimento de Software

Trabalho Prático
Grupo 15

Fábio Mendes, 8170157
José Baltar, 8170212
Rodrigo Coelho, 8170282

Índice

Índice de Figuras

1. Introdução

1.1. Apresentação

Para a execução e desenvolvimento deste projeto, incluído no âmbito da Unidade de Curricular "Laboratório de Desenvolvimento de Software" e de tema livre à escolha do grupo, foi escolhido o desenvolvimento de um de um videojogo de estratégia multijogador, do estilo *Tabletop*, no qual quatro jogadores poderão competir entre si até que exista apenas um vencedor.

Neste relatório irão ser explicadas as decisões e mudanças tomadas neste novo ciclo de desenvolvimento, assim como as tarefas de implementação do software anteriormente desenhadas na primeira *milestone*.

1.2. Contextualização

Na segunda *milestone* do projeto, foi decido dar prioridade à implementação inicial da lógica de jogo e comunicação com os clientes. Desde então, as tecnologias a serem utilizadas e a sua estrutura básica manteve-se, embora a lógica do jogo, tal como estava planeada, sofreu alterações e mudanças, pelo que alguns dos requisitos delineados na primeira *milestone* sofreram algumas modificações. Estas modificações são tanto visíveis no atualizado documento de *SRS*, como no GitLab do projeto.

Nesta entrega foi então desenvolvida a ligação entre o servidor e os diversos clientes que a ele se irão juntar, e criada grande parte da lógica base que torna o jogo jogável tanto no cliente como no servidor. Como descrito, os métodos lógicos encontram no servidor, pelo que o cliente tem a responsabilidade de mostrar aos jogadores o novo estado de jogo derivado das suas ações, isto é, um estado atualizado do jogo. Adicionalmente foi já criado o *frontend* da plataforma que irá servir de "GameClient" em browser.

1.3. Conceitos do jogo atualizados

Neste tópico são abordados os principais conceitos que fazem parte da estrutura do jogo.

Começando pelo mapa, este terá sempre uma forma fixa e Hexagonal com três lados maiores e três lados mais pequenos. Três fações (jogadores) localizadas nos lados mais pequenos, com outra fação centrada no mapa. O terreno entre as diversas fações é neutral e poderá ser anexado por estas, até se tornar o centro do confronto.

Existem, portanto, quatro Fações que representam os quatros jogadores de cada partida, sendo estas e as suas características:

- Remnant-> Cor Vermelha, centrada no centro do mapa, Ponto forte: Defesa;
- Confederation-> Cor Amarela, Sul do mapa; Ponto forte: Defesa;
- Royalists-> Cor Azul, Nordeste do mapa; Ponto forte: Ataque;
- Hordes-> Cor Verde, Noroeste do Mapa; Ponto forte: Ataque;

Em relação aos Recursos disponíveis, existem quatro que podem ser gerados e utilizados: Ouro e Metal (Minas), Madeira (Serraria), Comida (Quintas) e Manpower (Cidades). Os recursos são gerados pelas diversas regiões que o jogador ocupa. Devido a esse facto, certas regiões são especializadas em tipos de recursos diferentes, sendo que apresentam diferentes bónus dependendo do tipo de região. Os recursos são usados unicamente para criar e manter unidades de tropas e exércitos.

A nível diplomático, cada Fação encontra-se, inicialmente, em paz com as restantes. É através de declarações de guerra que a guerra começa, quando as fações criam fronteiras possíveis umas com as outras. Paz pode ser declarada apenas 1 vez para cada fação. Alianças oficiais entre jogadores não são possíveis.

Existem vários tipos de tropas, todos eles disponíveis a todas as fações, sendo estas:

- Infantaria Pesada
- Infantaria Ligeira
- Míssil
- Cerco
- Cavalaria Pesada
- Cavalaria Ligeira
- Cavalaria Míssil

As fações lutam através de Exércitos, compostos por diversos tipos de tropas, até um máximo de 15 unidades. Para se criar um exército é necessário usar os recursos disponíveis, numa taxa fixa. Para se treinar tropas, deverá estar estacionado numa região em que o recrutamento é possível. Os exércitos têm uma *manpower* associado a cada unidade. Esse *manpower* é o total de "tropas", ou equivalente a um HP e a Força, simultaneamente. Em combate, este será drenado com base no resultado dos exércitos. Para ser reabastecido, o exército terá de ser “reparado” para recrutar novas tropas para unidades. Cada região apenas pode ter um exército estacionado. Um exército só se poderá mover para uma região diretamente vizinha e sem barreiras ou outros exércitos estacionados. É possível atacar uma região, vizinha, caso tenha exércitos estacionado, combate será iniciado. Caso não tenha, a região é anexada sem qualquer custo.

Cada Região tem um tipo de relevo diferente, podendo assumir os valores de "Montanhoso", "Acidentado" (colinas) ou "Plano". As diferentes unidades têm bónus diferentes para cada tipo de terreno, que afeta a sua capacidade de poder máxima.

1.4. Entregas e *Milestones*

Todo o desenvolvimento do projeto consolidou-se em três eventos principais, cada um marcando um momento de entrega relativamente ao estado do trabalho à data. Resumidamente, em cada um desses momentos fizeram-se as seguintes entregas:

- Para a primeira entrega fez-se uma análise de opções e abordagens para o começo e seguimento do projeto. Ficou concluída a descrição de uma arquitetura inicial do e toda a documentação dos primeiros requisitos das aplicações.
- Na segunda entrega, seguiu-se com a implementação do *core* do projeto, o serviço web do motor de jogo, mais o sistema de gestão de salas e procura por partidas. Da mesma forma, deu-se início à implementação do cliente de *renderização* em *Unity* e o planeamento do cliente de jogo no Browser.
- Numa última entrega, ficou concluída toda a implementação para o cliente de jogo no Browser, tal como para a API de armazenamento de dados sobre utilizadores, histórico de partidas e notificações. Igualmente, finalizaram-se todas as atualizações necessárias aos sistemas desenvolvidos anteriormente, como o sistema de salas e todas as restantes funcionalidades do jogo, tanto no servidor como no cliente.

2. Arquitetura

2.1. Descrição dos componentes do projeto

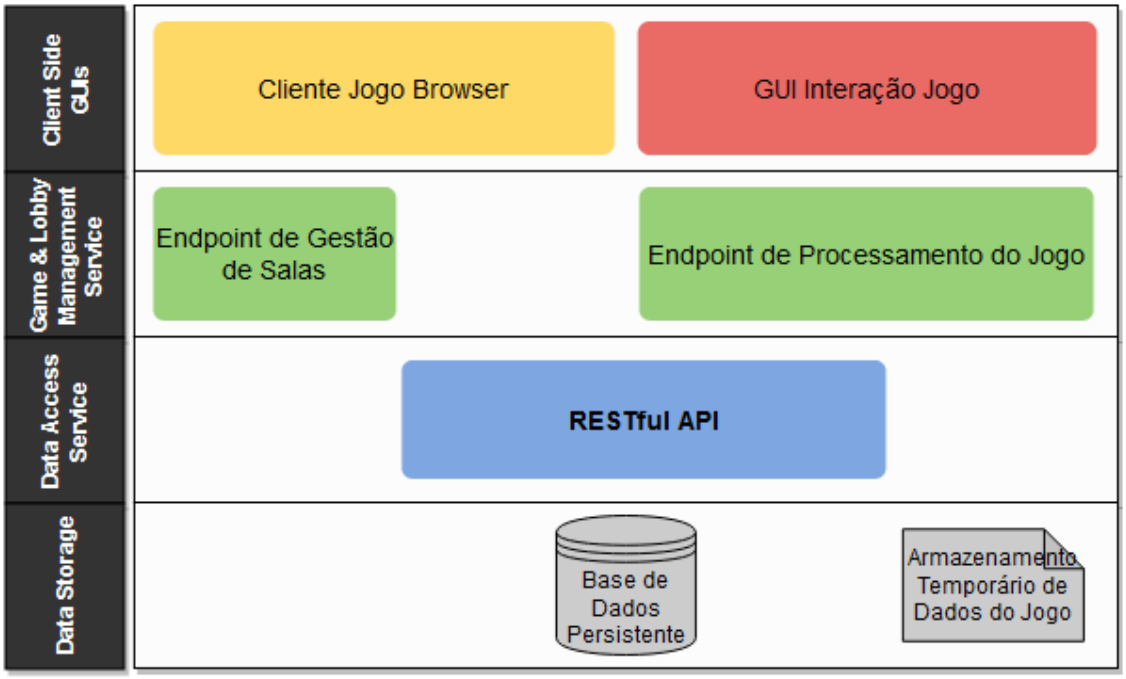


Figura 1 - Arquitetura Conceptual do Projeto

Como versão final, a arquitetura assenta num Serviço Web (ASP.NET Core) responsável pelo armazenamento e gestão de toda a informação relativa ao Sistema, desde dados de autenticação e utilizadores até ao histórico de jogos. Comunicam com o serviço tanto os Clientes, descritos mais à frente, como o Servidor Dedicado de jogo.

Então, sendo o jogo um multiplayer de estratégia, deverá existir uma ferramenta para que os jogadores se consigam organizar entre si, ou automaticamente, numa *sala* antes de dar início ao jogo. Sendo assim, foi decidido o desenvolvimento de um *Game Client* em browser para o efeito. Este *Game Client* será responsável por disponibilizar aos Utilizadores uma interface de acesso aos seus dados pessoais, histórico de jogos, autenticação e ao sistema de *salas* ou *matchmaking*. Um segundo Cliente será a própria interface do jogo, desenvolvida em Unity, que fornece todos os mecanismos de interação gráfica com as mecânicas do jogo, sendo que todo o processamento se irá realizar no próprio servidor.

Finalmente, a comunicação entre os jogos dentro do servidor dedicado será feita através de Sockets. E sendo que uma *sala* em princípio deverá conter um sistema de chat, também serão utilizados Sockets neste contexto. Ou seja, o sistema de *salas* e o servidor dedicado ao jogo estarão contidos num mesmo serviço, desenvolvido à base de Sockets.

2.2. Autenticação e Autorização

Para garantir privacidade e consistência em todo o projeto, existirá um processo de autenticação que permite ao utilizador manter a sessão iniciada entre qualquer componente, desde o Login no browser até ao início do jogo no Desktop. Igualmente, no final de cada partida o servidor de jogo envia os dados de histórico para a API, ficando também necessário existir alguma forma de autorização do servidor para determinados endpoints da API.

No contexto de autenticação e sessões após o *sign-in* de utilizadores, foi integrado no projeto a utilização de tokens JWT. No fundo, o token será verificado em cada pedido realizado à API, tal como em cada nova conexão com o servidor de gestão de salas e processamento do jogo. Então:

- O serviço Web estará disponível para registo de novos utilizadores e realização de *sign-in*, onde será gerado um JWT token. Cada pedido HTTP após o *sign-in* será autenticado através do JWT token enviado pelo *header* de autenticação.
- Por sua vez, o servidor de gestão de salas e processamento do jogo espera por novas conexões. Quando um cliente se conecta é enviado o JWT token, relativo à sua sessão, em forma de cookie, pelo pedido HTTP de transição para o protocolo WebSocket. O Servidor então comunica com a API para confirmação da autenticidade do token. Sendo válido, o servidor estabelece o protocolo de comunicação com o cliente, caso contrário termina a comunicação.

Para mais detalhes, está presente no documento SRS, em anexo, um diagrama de sequência UML que descreve a explicação acima de forma mais visual.

Relativamente à comunicação entre o servidor do jogo e a API no final de cada partida, a autorização é processada através de uma *string* especial enviada em cada pedido e verificada na API. A ideia é que o servidor do jogo seja o único cliente capaz de adicionar novas entradas de histórico de partidas para os jogadores. Seria possível seguir uma abordagem diferente, como a criação e uma conta de administrador especial, contudo viu-se desnecessário visto o curto tempo para a entrega do projeto, tal como o facto de ainda se continuar em ambiente de desenvolvimento.

É também importante notar que, no cliente, a interação com o jogo está completamente separada da interação com as salas. Contudo, esses dois componentes comunicam com um mesmo serviço. Desta forma, nesse serviço estão dispostos dois *endpoints* diferentes. Ou seja, quando um cliente faz ligação ao servidor pelo jogo já deverá estar identificado numa sala que

deu início a esse jogo. É assim necessário existir uma identificação única em cada sala para cada jogador, tal como ser possível relacionar a conexão, em *socket*, ao jogador.

2.3. Tecnologias para cada componente

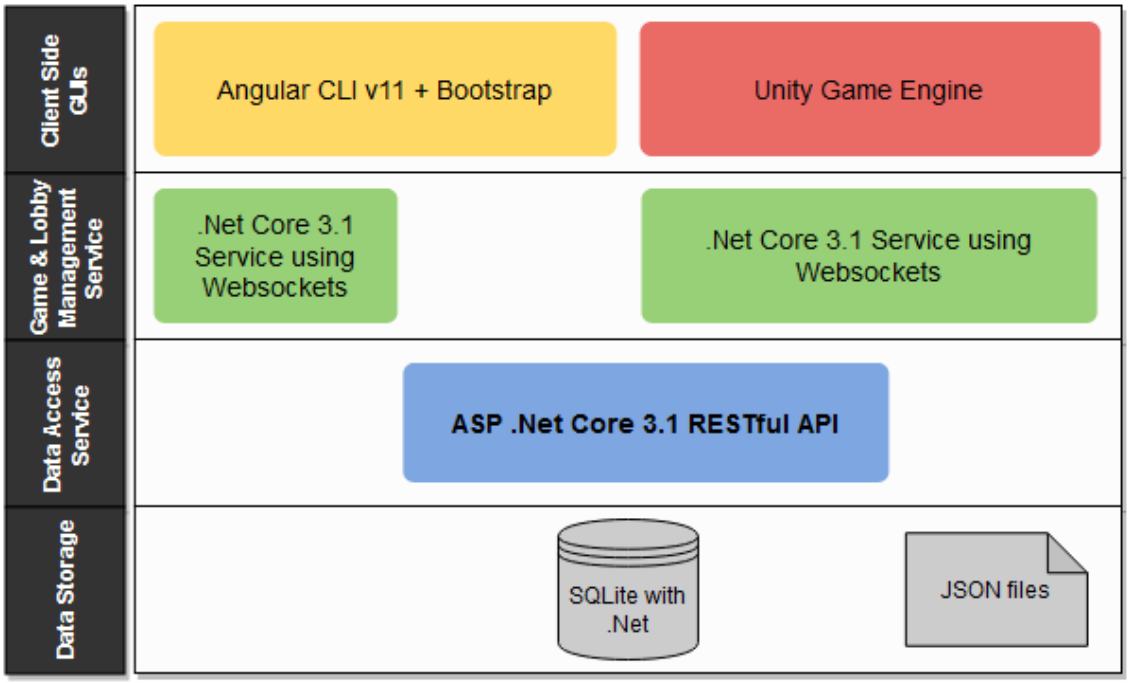
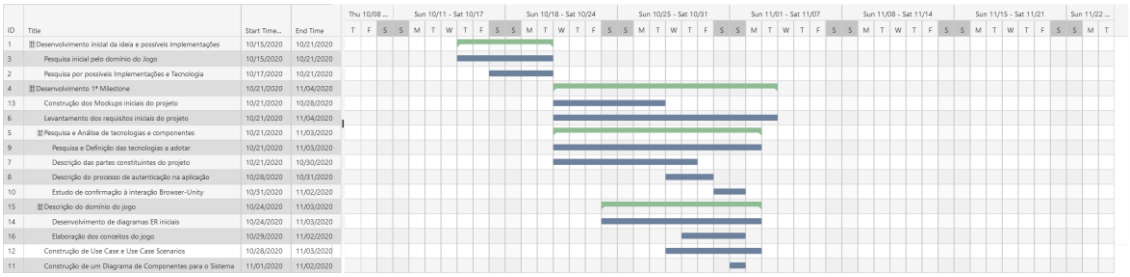


Figura 2 – Tecnologias para cada Componente

Adicionalmente, foi realizado um estudo com o intuito de confirmar que é possível iniciar a Interface de Jogo, desenvolvida em Unity, pelo do Browser. Este processo pode ser visto em maior detalhe no **Anexo-A** deste documento.

3. Primeira Milestone

3.1. Planeamento da Primeira Entrega



ID	Title	Start Time	End Time
1	Desenvolvimento inicial da ideia e possíveis implementações	10/15/2020	10/21/2020
3	Pesquisa inicial pelo domínio do Jogo	10/15/2020	10/21/2020
2	Pesquisa por possíveis Implementações e Tecnologia	10/17/2020	10/21/2020
4	Desenvolvimento 1ª Milestone	10/21/2020	11/04/2020
13	Construção dos Mockups iniciais do projeto	10/21/2020	10/28/2020
6	Levantamento dos requisitos iniciais do projeto	10/21/2020	11/04/2020
5	Pesquisa e Análise de tecnologias e componentes	10/21/2020	11/03/2020
9	Pesquisa e Definição das tecnologias a adotar	10/21/2020	11/03/2020
7	Descrição das partes constituintes do projeto	10/21/2020	10/30/2020
8	Descrição do processo de autenticação na aplicação	10/28/2020	10/31/2020
10	Estudo de confirmação à interação Browser-Unity	10/31/2020	11/02/2020
15	Descrição do domínio do jogo	10/24/2020	11/03/2020
14	Desenvolvimento de diagramas ER iniciais	10/24/2020	11/03/2020
16	Elaboração dos conceitos do jogo	10/29/2020	11/02/2020
12	Construção de Use Case e Use Case Scenarios	10/28/2020	11/03/2020
11	Construção de um Diagrama de Componentes para o Sistema	11/01/2020	11/02/2020

Figura 3 - Diagrama planeamento da primeira entrega

3.2. Criação e Desenvolvimento de Diagramas e outros Sistemas

Foram elaborados alguns diagramas, com o presente propósito de explicar, uma forma simples as diversas componentes do projeto.

Os diagramas ER criados procuram explicar as diversas relações entre as entidades que irão permanentes e estarão guardadas de forma a suportar a lógica de funcionamento dos diversos sistemas e aquelas que descrevem a situação em que estes se encontram naquele momento. Ou seja, são estes os dados em específico que os componentes necessitam de forma a poder funcionar e modela de forma bastante inicial a forma como certas componentes de cada um dos sistemas funcionam.

Relativamente ao sistema de comunicação e componentes globais do projeto, são de notar alguns dos *mindsets* seguidos durante a pesquisa desenvolvida durante a primeira milestone:

- Inicialmente a ideia do "Game Client" não existia, ou seja. seria tudo desenvolvido dentro do motor do Unity. Esta seria uma ideia igualmente viável, no entanto decidimos seguir com a versão do browser por ser mais enriquecedor e multiplataforma. Um utilizador poderá assim aceder ao seu histórico de jogos através de qualquer máquina que contenha um Browser, como um smartphone, contudo não poderá entrar numa "sala" para dar início a um jogo.
- Também se pensou em desenvolver o sistema de salas através do serviço HTTP, mas isto impossibilita ou tornaria complicado a adição do sistema de chat dentro das salas, tal como a transição da sala para uma instância de jogo no servidor dedicado.

Sobre o sistema de autenticação, outras ideias pertinentes pensadas pelo grupo na altura da primeira milestone, mas que não conseguiram o *final* cut, foram:

- Foi considerada a alternativa de fazer a autenticação do cliente na API e nos servidores de jogo no *mesmo local*. No entanto, sendo que a implementação dos dois serviços difere bastante (HTTP requests e TCP/UDP sockets, respetivamente) ficou decidido que deverão estar separados. Conclui-se então que a autenticação é realizada sempre pelo serviço web.

3.3. Criação de Requisitos e Use-Cases

O principal esforço desta entrega consistiu precisamente na construção de Requisitos funcionais. Os requisitos desenvolvidos consistem no entendimento presente acerca daquilo que planeamos para o jogo e também para o que cada utilizador poderá fazer e consultar. Com isto, os requisitos apresentam a forma como pretendemos que o jogo funcione de uma maneira mais prática, isto é, além de meros conceitos, apresentando um conjunto de características e condições que são de fácil entendimento e possível transcrição para a programação do próprio jogo e também de todas as componentes do cliente, tais como *matchmaking*, entre outros. No entanto, estes requisitos não são necessariamente os finais. Uma vez que estamos a trabalhar utilizando metodologias ágeis, estes servem como *kickstart* para o entendimento geral do projeto, mas são passíveis a mudança e a incremento de novos requisitos, tendo em conta o desenvolvimento do projeto e gestão de mudança que poderá ter de ser efetuada.

Em relação a Use-Cases e a Scenarios, pretende-se mais uma vez mostrar funcionalidades do projeto, mas desta vez da perspetiva do utilizador. Ao longo do correr do projeto prevê-se que os use-cases scenarios sejam alterados principalmente em relação aos passos a percorrer para executar a funcionalidade, visto que nesta fase do projeto ainda não temos uma ideia muito solidificada relativamente à interação do jogador com o jogo.

3.4. Desenvolvimento de *Mockups*

Ao longo do desenvolvimento do projeto foram desenvolvidos mockups, com o principal objetivo de ajudar a ilustrar as potencialidades do jogo e das interações entre os jogadores e a interface. Os mockups desenvolvidos correspondem, sobretudo ao GameClient (estilo BattleLog) em browser e à UI de Jogo, embora não sejam, de qualquer maneira, finais ou indicativas do estado final do produto, uma vez que complexidades de implementação ou outras particularidades tornam uma implementação exata e adequada destes mockups difícil e até desnecessário.

4. Segunda Milestone

4.1. Planeamento

Ao contrário da milestone anterior, onde foi construido um diagrama de Gant para o planeamento, todo o planeamento desta segunda milestone está descrito em dois *Sprints*, detalhados no repositório GitLab relativo ao projeto do grupo.

4.2. Comunicação e Websockets

Partindo do princípio de que o nosso jogo é um Strategy Game que funciona à base de timers e decisões, sendo o tempo de reação quase inexistente, não será importante dar muita atenção a grandes implementações para gestão de *Lag*. Contudo, isto não significa que se vá ignorar o tema por completo! No fundo, o *lag* será gerido pela forma em que os dados serão enviados entre os jogadores e não tanto pela qualidade das conexões.

Então, visto que a implementação do servidor dedicado do jogo e gestão de conexões é concretizada como um serviço web, concluiu-se que será utilizado o protocolo WebSockets para toda a comunicação client-server em tempo real, em que múltiplos clientes se conectam a um único servidor que fornece os serviços necessários. Contudo, o grupo tem consciência de possíveis problemas relativamente a este tema.

WebSockets é um protocolo que junta um pedido HTTP com a abertura de um canal em TCP, para construir uma *ponte* de comunicação bidirecional entre o cliente e o servidor. Em TCP, o maior problema que o grupo poderá encontrar neste contexto será o bloqueio da comunicação em caso de packet loss, já que o protocolo se encarrega da organização dos pacotes enviados. Isto significa que a utilização do protocolo TCP deverá existir em situações como o envio dos inputs do jogador para o servidor e mensagens de chat, sendo que o envio de broacasts se pode tornar problemático, contudo o grupo ainda não encontrou problemas com tal.

É de notar também que as mensagens enviadas entre as conexões estarão descritas em formato JSON. Ou seja, para cada mensagem enviada existe *serialization* do lado de quem envia e *deserialization* por quem recebe. É possível que isto venha a criar alguma lentidão no envio de cada mensagem, contudo é a melhor abordagem a seguir para o momento, tendo em conta tudo o que já foi e será descrito neste relatório.

Falando em específico do contexto do jogo, e como pode ser visto pelo diagrama abaixo, a comunicação tem foco principal a partir do Cliente, uma vez que cada Cliente envia informação sobre os eventos que irá efetuar a cada turno de jogo. O Servidor tem então a tarefa de os

executar e, a cada final de turno, enviar informação atualizada para todos os clientes conectados àquela sala.

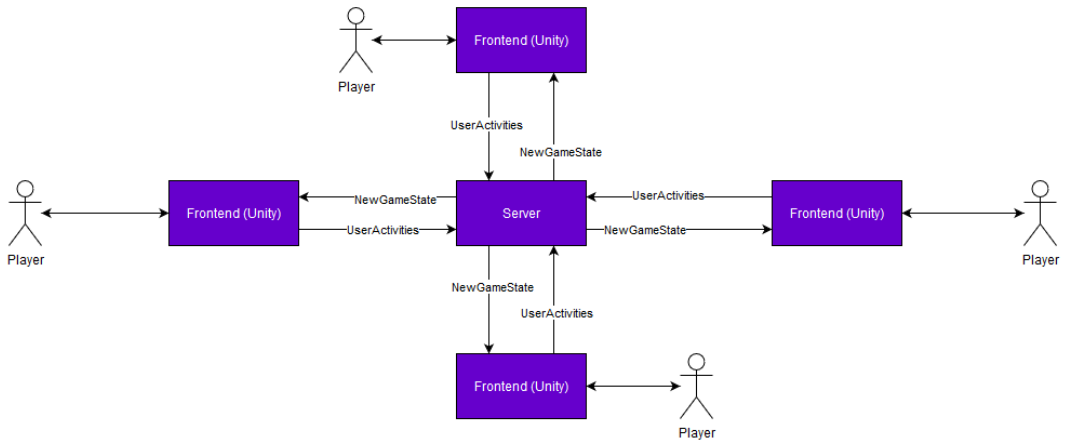


Figura 4 – Comunicação para processamento e atualizações do Jogo

4.3. Lógica de jogo e serviço web correspondente

Para a segunda *milestone*, foi importante haver já um foco grande na lógica de jogo, em servidor, uma vez que seria o core do projeto enquanto produto. Foi, por isso, considerado crucial que se começasse pelos métodos que efetivamente formam o conjunto mais fulcral do projeto, que nós consideramos como a "Lógica de Mapa", ou a Gestão de Mapa.

De um ponto de vista programático, este Mapa definido no servidor é um Grafo implementado com recurso a uma Matriz, sendo gerado através da leitura de um ficheiro em formato JSON contendo as informações referentes a cada região, isto é, a unidade atómica do mapa. Cada vértice é, portanto, uma ligação do grafo e a lógica de mapa é uma iteração baseada em princípio de abstração sobre este facto. O grafo de mapa é um Grafo Não Pesado e Não orientado, contendo apenas as ligações lógicas, ou seja, as fronteiras, entre as diversas regiões.

Além do Mapa, foi também considerado crucial o desenvolvimento referente às componentes para Exército, Unidades e Fações, incluindo a gestão de recursos disponíveis a cada fação. Assim, sendo, foram implementados os métodos que permitissem a que estes interagissem entre si e com as regiões já previamente implementado no 1º Sprint. Utilizando a Coleção "List" existente em C#, um gestor para a Instância de jogo, que guardará todos os diversos dados referentes ao presente estado do jogo no servidor são criados. Tal como descrevem os requisitos, foram implementados métodos que potenciem a criação de exércitos, recrutamento de unidades para estes, movimento, combate, anexações e conquistas de região, além de métodos mais específicos que se encontram detalhados nos requisitos do projeto, *no*

documento SRS anexo. *Detalhes sobre o seu funcionamento são encontrados no mesmo documento.*

Como forma de tratar a informação recebida dos clientes, o servidor está sempre "à escuta" de "Eventos" (anterior designados de pedidos) causados pelos jogadores, eventos esses que guarda numa Queue, ordenando de forma natural pela ordem em que chegam ao servidor. Esses pedidos são depois encaminhados para um **"Protocolo"** que separa e executa as ações que são necessárias para cada específico evento, chamando o método designado na Instância de jogo. *Detalhes sobre o protocolo mencionado podem ser analisados no documento SRS em anexo*.

O que acontece é que, do ponto de vista de servidor, o estado de jogo poderá estar em constante atualização, no entanto, do ponto de vista do cliente, as coisas estão se a manter iguais durante o período de duração do turno. Por isso mesmo, garantimos que desta maneira não existem "conflitos" uma vez que todas as ações são executadas com um certo *flow*. O que acontece é que certas ações que o jogador não conseguiria antever poderão, de facto, acontecer, uma vez que não está a verificar o conjunto de ações que os outros jogadores estarão a tomar. Uma vez que estas ações são inesperadas, significa que um conjunto de atributos poderá ter sido completamente alterado no lado do servidor, o que compromete a integridade da solução. Para tal, um conjunto forte de verificações é aplicado para garantir que os dados ainda se encontram inalterados. Caso haja operações que não estejam necessariamente dependentes destes dados, é necessário tornar a sua execução o mais generalizada possível. Em casos menos extremos, o que poderá acontecer é que uma certa ação será bloqueada.

De um ponto de vista estrutural, a solução foi implementada recorrendo a um conjunto de "Managers" que tratam de cada componente em específico, guardando em Lista as informações pertinentes ao Estado de jogo. Quando um evento é chamado no Protocolo, a Instância de jogo pertencente ao Room, à qual a informação é enviada, conforme previamente mencionado.

4.4. *Client-side* Unity e Integração com o serviço web

No cliente Unity, houve um grande foco no sistema de comunicação com o servidor responsável pela gestão do jogo e na interação do jogador com a interface.

Relativamente a comunicação com o servidor foram implementados duas threads, uma responsável por receber os dados do servidor e outra responsável por enviar os inputs dos utilizadores para o servidor. Foi implementado deste modo para que estas tarefas não interferiam com a performance do jogo, ou seja, que não atrasem a execução de outras operações. Com esta implementação não se pode colocar a thread responsável por receber os dados do servidor também responsável por processá-los e atualizar o jogo em si, porque existem

certas ações do Unity que não são “thread-safe”. Com isto em mente a thread em questão apenas recebe a informação e coloca-a numa ConcurrentQueue que outros objetos podem aceder e por sua vez processar a informação.

Também é importante salientar que a Classe responsável pela com comunicação com o servidor é um singleton, isto quer dizer que a instância da classe é a mesma onde quer que seja acedida na “Scene” do jogo.

Quanto à interação do utilizador com a interface foram impostas as devidas restrições para impedir o congestionamento do servidor com pedidos repetidos ou pedidos que já se sabe que o servidor vai “recusar”. Por exemplo quando se cria um exército numa região é verificado se o utilizador é o dono, se já não contem um exército, o tipo de região, se já não colocou um exército a mover-se para essa região na mesma ronda, entre outras restrições.

Outro caso é a criação de “dummies”, e por “dummies” quer-se dizer objetos que ainda não foram processados pelo servidor, mas que são necessários para o utilizador poder executar outras ações que os envolve. Como no caso de criar um exército, esse input só vai ser processado para a próxima ronda, mas na ronda em que é criado o utilizador já pode adicionar unidades a esse exército, isto também bloqueia a opção de criar exército nesta região o que, como referido anteriormente, previne o congestionamento do servidor. Neste caso é necessário criar um “dummies” tanto para o exército como para a as unidades adicionadas.

Estes “dummies” quando se recebe o update são eliminados e substituídos pelo objeto “real”, o que é recebido pelo servidor.

Nesta milestone o cliente é que está responsável por pedir as atualizações ao servidor, mas no próximo milestone este comportamento irá ser alterado e o servidor é que ficará responsável por enviar os updates para os clientes.

4.5. *Client-Side* Angular e serviço web de Gestão de Salas

Toda a implementação no *front-end* para o componente *Game Client* está desenvolvida recorrendo à framework Angular. Igualmente, para garantir que a implementação respondesse aos requisitos do domínio, foram construídos mockups como representação base do produto final. Contudo, estes mockups têm como objetivo demonstrar visualmente as funcionalidades requeridas para o projeto e não tanto dar como final a aparência cosmética, ou seja, cores e alguns posicionamentos do conteúdo poderão sofrer modificações durante a implementação.

No entanto, é de notar que nem todas as funcionalidades deste componente ficaram planeadas para esta milestone, visto que a implementação da API de armazenamento e acesso aos dados dos utilizadores está planeada para a terceira milestone. Com isto, do ponto de vista

funcional o *Game Client* atualmente contém implementado toda a organização das páginas web, a inicialização de *lobbies* e, em parte, a transição para o jogo, sendo que esta última terá que ser manualmente ativada. *Mockups e outros detalhes podem ser vistos no documento SRS em anexo.*

Então, para que seja possível o relacionamento entre salas de lobby criadas pelo componente *Game Client*, descrito acima, e salas relacionadas ao processamento do próprio jogo, está implementado no servidor um sistema capaz de gerir um conjunto de salas de forma concorrente. O sistema é utilizado inicialmente pelo endpoint que gere as conexões vindas do cliente no browser e depois acedido pelo endpoint de gestão de conexões vindas do cliente de interação com o jogo, em Unity.

Resumidamente, a lógica do sistema passa pela partilha de uma coleção de *salas* entre todas as conexões ligadas a qualquer um dos endpoints. Para isto ser possível, foi seguida uma abordagem baseada no conceito de *Singleton*, uma classe que contém e gere a referida coleção e é instanciada uma única vez. Dito isto, é absolutamente necessário existir um controlo de acesso concorrente aos dados, visto que a cada nova conexão, atualização ou fecho essa coleção sofre algum tipo de modificação.

Outro ponto importante a notar é o conjunto de verificações relativamente a cada sala em si. Isto é, cada sala terá no máximo até quatro jogadores. A transição entre o *lobby* e o *jogo* implica que seja necessário existir algum tipo de relação entre as duas salas. Para resolver este problema, cada instância de uma sala é a mesma para um *lobby* como para um *jogo*. No fundo, quando um jogador passa de um *lobby* para o *jogo*, após o evento de início de jogo, a sua instância no servidor, guardada dentro da sala, mantém-se e só se remove a referência do socket que representa a sua conexão. Após o início do jogo, o jogador volta a conectar-se e o servidor fica responsável por atualizar a nova referência dentro da sala onde o jogador estiver.

P*ode ser visto no documento SRS, em anexo, um diagrama de classes que descreve todo o Sistema de Gestão de Salas em maior detalhe.*

Concluindo, tendo em conta a abertura de um canal constante de comunicação entre clientes, em lobby, e o servidor é inevitável a criação de um sistema que organize as mensagens trocadas entre os dois lados. A ideia aqui será um seguimento do trabalho desenvolvido para o protocolo de comunicação entre os clientes em jogo, neste caso entre os clientes em lobby, no browser, e o servidor. Este **protocolo** tem como objetivo definir um conjunto de regras para as mensagens JSON trocadas, definindo no fundo um conjunto de *eventos* que representam operações com inputs e outputs. *Para mais detalhes sobre que tipos de eventos estão definidos no protocolo, veja-se o documento SRS em anexo.*

4.6. Planeamento e incrementos para a próxima Milestone

Para a próxima *milestone*, o planeamento preliminar implica que os métodos e processos começados nesta Milestone sejam efetivamente terminados e otimizados, além de abordar todos os outros requisitos que ainda necessitam de ser implementados.

Assim sendo, **para a próxima *Milestone***, o principal foco será de facto terminar todos os métodos referentes às vertentes do jogo, tanto no Cliente como no Servidor, assim como a implementação dos métodos que permitam a utilização da Plataforma que faça o suporte ao jogo. No entanto, é de esperar que se implementem alterações que corrijam qualquer tipo de problema, mau planeamento ou má execução de etapas da Segunda *Milestone*.

5. Terceira Milestone

5.1. Planeamento

Como na *milestone* anterior, todo o planeamento para esta última *milestone* ficou descrito em três *Sprints*, detalhados no repositório GitLab relativo ao projeto do grupo.

5.2. Desenvolvimento continuado do jogo em Servidor

Uma vez que os métodos *core* do jogo já se encontravam desenvolvidos e verificados, em larga medida, na Segunda Milestone, o desenvolvimento do jogo para esta *milestone* acabou por ser um pouco mais curto. Como tal, uma das prioridades foi a automatização das componentes referentes ao turno, à colmatação e correção de problemas e melhoria da lógica de jogo, além dos incrementos necessários frente aos requisitos que nos foram apresentados. A arquitetura e forma de comunicação mantiveram-se da mesma forma, seguindo os preceitos utilizados na segunda milestone.

Descrevendo exclusivamente aquilo que foi adicionado à lógica do jogo em servidor, esta Milestone consistiu na adição dos métodos referentes à Vitória, Derrota (seja através de condições de jogo automática, ou por capitulação) e à Finalização de um determinado jogo, incluindo o historial de jogo, o que incluí a ligação à *RESTful API* do Servidor Web enviando um pedido POST.

5.3. Conexão, Sistema de Turnos e Verificação

Quando um jogador se conecta ao jogo, este envia um pedido/evento de "InitialInformation", o que informa o utilizador que já se encontra conectado e pronto a receber mensagens para desbloquear o jogo. Quando o primeiro jogador se conecta ao jogo, é iniciado um temporizador de sessenta segundos, em que se espera pela entrada de mais jogadores na partida. Assim que todos os jogadores estejam conectados, ou o temporizador acabe, o jogo é dado como iniciado e é enviada a informação inicial para todos os jogadores que se encontrem conectados. A informação de quais jogadores se encontram conectados é guardada num dicionário base do C#.

O jogo utiliza um sistema de turnos automatizados, referidos informalmente como "Turnos Dinâmicos", uma vez que os jogadores não têm controlo sobre a sua duração. Um turno é tido como a unidade básica do tempo, com uma duração fixa de 60 segundos, nos quais os seus primeiros 15 segundos os jogadores não podem acionar nenhum evento do jogo. Também, a cada 20 segundos, as ações efetuadas pelos diversos clientes até aí são executadas, o que

procura aliviar um bocado a pressão sobre o servidor quando o turno acabar, fazendo com que existem 2 momentos anteriores em que cerca de 25 segundos de interação são tratados antes do seu término, deixando apenas 20 segundos adicionais que serão tratados quando o turno acabar. Os turnos são simultâneos para todos os jogadores, ou seja, as suas ações decorrem concorrentemente e a sua execução está dependente da ordem *FIFO*, *First In First Out*, utilizando uma *ConcurrentQueue* nativa ao C#, uma *queue* especializada que permite o acesso por diversas threads de uma maneira concorrente mais segura e eficaz. Esta é uma diferença relativamente ao que fora realizado anteriormente, uma vez que era utilizada uma *Queue* desenvolvida para o efeito.

Este método concorrente cria alguns problemas lógicos, uma vez que certas ações poderão invalidar outras. Desta forma, é necessário um conjunto de verificações fortes que permitam a existência de um estado de jogo em servidor que seja coerente com os eventos e ações acionados e executadas em clientes. Por exemplo, uma vez que um exército a qualquer dada altura poderá ser atacado e destruído, é necessário verificar, em todas ações envolvendo exércitos, se este ainda existe e nas condições descritas.

Este método permite, portanto, a existência de ações concorrentes e manter a integridade lógica das diversas ações, ainda que possa introduzir, do ponto de vista do jogador, algumas ações que são inesperadas, mas naturais tendo em conta a natureza do jogo e também do tema em que é inspirado.

Um grande foco desta entrega foi precisamente garantir a fiabilidade da receção e transmissão entre todos os componentes assim como a máxima sincronização de todas as ações.

5.4. Derrotas das Fações e Término do Jogo

Como será lógico, as fações deverão ser derrotadas a dada altura durante o decorrer do jogo, associadas à sua possessão sobre a sua capital. Estas opções de derrota são verificadas no final de cada turno, em que se verifica o estado de cada fação e a sua relação com a sua capital, ou seja, se esta ainda mantém possessão sobre esta. Caso uma fação seja derrotada, esta é marcada como “defeated”, em que *defeated* é uma variável booleana em que o valor *true* indica que essa fação já foi derrotada). O cliente verifica, então, se esta fação já foi derrotada e se foi, bloqueia as ações do cliente, assim como o servidor, que deixa de aceitar eventos de uma fação que foi já derrotada, deixando passar apenas mensagens de chat, que também estão bloqueadas do lado do cliente. Além disso, uma fação poderá capitular, saindo do jogo e marcando a sua fação como derrotada. Quando uma fação é derrotada, as suas estatísticas do jogo são guardadas numa Collection (List) nativa do C#, associada ao GameRoom em que os jogadores

presentemente se encontram. O jogador poderá então sair da sala a qualquer momento, e a sua presença na sala será limpa, permitindo que se conecte a qualquer outro jogo, ou poderá ficar a observar o resto da partida até ao seu término.

A cada turno, o jogo verifica se já existe um vencedor. O vencedor é detetado verificando quantas fações já foram derrotadas dentro da sala, isto é, se já foram 3 fações derrotadas ou capituladas, a última fação marcada como ativa é aquela declarada como vencedora. Com isto, o jogo é dado por terminado, as novas informações de estado do servidor são mandadas para os clientes e processadas lá, indicando que o jogo já se encontra terminado. Portanto, podemos dizer que um jogo termina quando um jogador/fação sai vitorioso do confronto. Do lado do servidor, o GameRoom é removido do servidor e os jogadores conectados na altura poderão então ingressar numa outra partida. É também nesta altura que os dados da partida, para cada jogador, são enviados, através de um pedido POST, para a RestAPI e guardados na base de dados SQLite aí contida. O utilizador poderá aí consultar o seu próprio histórico de partidas, ou também o histórico de outros jogadores.

5.5. *RESTful API* de armazenamento de dados

Toda a API foi desenvolvida em conjunto com as funcionalidades requeridas no lado do cliente. Ou seja, a API existe como resposta a funcionalidades e não tanto como fonte para novas funcionalidades.

No fundo, como o nome indica, esta aplicação foi desenvolvida para disponibilização de dados globais ao projeto guardados de forma persistente. Para isto, o serviço faz uso da biblioteca *Entity Framework* para toda a interação com o armazenamento dos dados em SQL. Como Base de Dados, inicialmente e para facilitar o ambiente de desenvolvimento, utilizou-se SQLite para o armazenamento com o plano de atualização para SQL Server num ambiente de produção futuro. No entanto, como descrito na conclusão deste relatório, o grupo nunca chegou a conseguir fazer *deploy* da aplicação, logo SQLite manteve-se como a base de dados ativa.

Toda a documentação da API pode ser vista diretamente na API através do *endpoint* da framework *Swagger*.

5.6. *Game Client* do jogo para o Browser

Globalmente, todo o componente *Game Client* ficou finalizado neste último milestone, com a exceção da pesquisa por perfis de utilizadores, somente acedido através da lista de amigos, e autenticação externa pela Google, sendo estas duas as funcionalidades ainda que importantes que não tiveram um *final cut* na aplicação.

A aplicação está descrita em grande parte no capítulo 4.2 deste documento, sendo que nesta última milestone foram implementadas atualizações no funcionamento de alguns componentes já implementados, contudo sem alterações de estrutura, tal como ficaram implementados os mockups restantes e as páginas web de interação com os dados na API, nomeadamente a lista de amigos, histórico de jogos e notificações.

Contudo, nesta última entrega é de notar a forma como foram implementados os serviços Angular de comunicação com o serviços web, a API e o servidor de jogo. Isto é, recorreu-se em grande parte à utilização do padrão *Observable* e a programação assíncrona através da biblioteca RxJs. Igualmente, foi dado um grande foco a manter

Atualizações no sistema de salas, rxjs e programação assíncrona com paradigma funcional. (diagrama de classes atualizado e diagrama de sequência novo, início da aplicação Unity, em anexo)

5.7. Desenvolvimento Game Client Unity

Relativamente ao desenvolvimento do jogo foram implementadas as funcionalidades em falta, nomeadamente o vitória, derrota e desistência. À semelhança das outras funcionalidades, estas estão restritas pelo que o client recebe do servidor.

Quanto à representação de ações para o utilizador foram tomadas decisões com base no tempo que tínhamos para implementar e a natureza do jogo em si. Uma das decisões foi representar no mapa os exércitos do jogador e o dos inimigos que fizessem fronteira.

Quanto a outras ações nomeadamente representar no mapa ataques, movimentação de exércitos, discutimos como iriamos representar isso e ficou decidido que em cada ronda durante o utilizador poderia consultar uma lista das ações que foram efetuadas tanto por ele como por inimigos que o afetasse na ronda anterior.

A alternativa a isto seria representações gráficas dos exércitos a mover-se e a combater. Por um lado, isto tornaria o jogo mais dinâmico e interessante, mas por outro lado estas animações teriam de ocorrer no espaço de 15 segundos iniciais da ronda (utilizador só pode verificar os updates) e isto podia tornar-se confuso para o jogador para alem que acabaria por lhe dar menos tempo para analisar o que aconteceu.

5.8. Atualização e Desenvolvimento dos sistemas de Autenticação

Durante este último milestone, foram implementados os vários sistemas de autenticação entre os jogadores no *Client*, a API de armazenamento de dados e autenticação e o servidor do Jogo.

Resumidamente, são utilizadas estratégias como *JWT signed tokens*, para autenticação dos jogadores, e um sistema personalizado para autorização do Servidor de Jogo com a API, na altura de envio de informação sobre partidas decorridas, através de uma *string* secreta enviada em cada pedido HTTP. Estes sistemas estão descritos em maior detalhe num capítulo anterior e em artefactos presentes no documento SRS em anexo.

5.9. Testing da Aplicação

Como é natural, é necessário testar os diversos componentes da aplicação, sobretudo a sua integração conjunta, ligando as quatro principais componentes.

Não foram utilizados testes unitários, pelos que todos os testes foram corridos utilizando vários métodos diferentes, incluindo testes externos em que partes do código eram testados numa aplicação à parte para verificar se o seu comportamento era o pensado/idealizado. Outros componentes, como por exemplo, a UI do jogo para o Cliente, só podiam ser testados manualmente em ligação ao servidor do jogo, pelo que era necessário criar um determinado fluxo de testagem que envolvia todas as parcelas do jogo. A WebAPI, no entanto, é de possível testagem utilizando a *framework Swagger*.

Para os testes do servidor, foi utilizada uma página desenvolvida para o efeito, que permitia a comunicação de clientes e salas com o servidor, sendo que era necessário colocar manualmente o id do GameRoom, o nome do jogador e a sua key de autenticação. A partir daqui, era possível mandar Eventos para o servidor e testar a sua funcionalidade e as respostas em JSON que o servidor retornava.

Os testes mais importantes encontram-se descritos num documento em Anexo.

6. Conclusão e *Deploy* do Projeto

Visto que a ideia global do projeto é a construção de um jogo multiplayer online, para quatro jogadores em simultâneo, com processamento em *server side*, o grupo tinha planeado o *deploy* dos serviços desenvolvidos para uma plataforma de *hosting*, como o Azure, mas tal não se verificou.

Contudo, provou-se o funcionamento do desenvolvido a partir de um ambiente "mais local", através da plataforma LogMeIn Hamachi, ficando os serviços *hosted* na máquina de um dos elementos do grupo e acessível através da VPN (Virtual Private Network).

Anexos

Anexo A – *Start-Up* do jogo a partir do Browser

Resumidamente, será possível executar deste modo se registarmos um *custom url protocol* no Windows. Podemos fazer este registo de forma manual, mas a ideia seria utilizar a aplicação externa "Inno" para gerar o instalador do jogo, nesta aplicação é possível alterar o script de instalação para que faça o registo do protocolo no Windows, no momento da instalação.

Relativamente à implementação nas tecnologias envolvidas é necessário da parte do browser criar um elemento html do tipo href em que o valor seria "(nomeDoProtocolo):(parâmetros)".

Figura 5 – Exemplo de como ficaria no html

Isto é um exemplo de teste no qual o parâmetro "GameCode" é estático, numa situação real seria um valor dinâmico que dependeria da sessão de jogo à qual se refere.

Do lado da aplicação desktop é preciso aceder aos argumentos da linha de comandos do processo atual para obter os parâmetros que o browser envia. Para testar isto foi feito uma aplicação em Unity básica cuja única função é mostrar os argumentos da linha de comandos.

Figura 6 - Este é argumento que a aplicação do Unity recebe enviado a partir da web

Depois de receber este argumento só temos de extrair a informação necessária, neste caso seria o GameCode.