UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

STANLEY SILVA SAMPAIO

UM FRAMEWORK EXTENSÍVEL PARA APOIO À TOMADA DE DECISÃO ESPACIAL

Ouro Preto
2026

STANLEY SILVA SAMPAIO

UM FRAMEWORK EXTENSÍVEL PARA APOIO À TOMADA DE DECISÃO ESPACIAL

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Dr. Tiago Garcia de Senna Carneiro

Ouro Preto
2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Stanley Silva Sampaio

Um Framework Extensível para Apoio à Tomada de Decisão Espacial

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 25 de Fevereiro de 2026

Membros da banca

Tiago Garcia de Senna Carneiro (Orientador) - Doutor - Universidade Federal de Ouro Preto
Rodrigo César Pedrosa Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto
Aline Norberta de Brito (Examinadora) - Doutora - Universidade Federal de Ouro Preto

Tiago Garcia de Senna Carneiro, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 25/02/2026

Documento assinado eletronicamente por **Tiago Garcia de Senna Carneiro**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 02/03/2026, às 17:31, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1062460** e o código CRC **358A3699**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Dedico este trabalho ao meu pai, Paulo, que foi e sempre será meu exemplo de persistência, motivação e objetividade.

Agradecimentos

Agradeço a toda minha família, por me fornecer a estrutura e educação necessária para chegar até aqui, em especial meu pai, Paulo Sebastião Sampaio (in memorian), meu irmão, Deyvid Silva Sampaio, e tia Flávia. À Monique, por todo carinho, companherismo e compreensão. Aos Amigos que Ouro Preto me proporcionou. Agradeço também à UFOP, em especial ao DECOM, pelo ensino de qualidade. Felizmente conheci várias mentes brilhantes que me ajudaram nessa trajetória, então agradeço a todos os professores e funcionários que tornaram possivel o ensino público de qualidade. Ao Professor Dr. Tiago Garcia de Senna Carneiro, por todo o suporte oferecido para a realização desse trabalho. Você é uma referência no contexto de Engenharia de Software, e seus ensinamentos, com certeza são essênciais para a evolução da àrea. À República UPA (Unidos por Acaso) por ter me acolhido ao longo desses anos, e ter garantido a experiência de conhecer pessoas que hoje chamo de família.

"Sistemas de software são complexos e devem ser projetados e construídos cuidadosamente."
– (SOMMERVILLE, 2011)

Resumo

Este trabalho apresenta o desenvolvimento de um sistema extensível para apoio à tomada de decisões geoespaciais no contexto do planejamento de viagens. Para isso, a plataforma Terraplanner, originalmente não extensível, foi adaptada a fim de suportar o carregamento dinâmico de plugins em seu front-end WEB. Como prova de conceito, o TerraTripper foi implementado como um plugin que permite ao usuário criar roteiros de viagem terrestres personalizados com base em parâmetros como tempo, custo, atrativos desejados e locais de hospedagem. A arquitetura de plugin desenvolvida promoveu modularidade, reuso de componentes e integração fluida com o sistema principal. O processo de desenvolvimento incluiu a elaboração de um Documento de Especificação de Requisitos (DER), a implementação de um MVP funcional e sua validação técnica. Os resultados indicaram a viabilidade da abordagem proposta, com potencial de expansão para novos domínios de apoio à decisão geoespacial.

Palavras-chave: Sistemas de Apoio à Decisão Espacial (SDSS). Plugins. Terraplanner. TerraTripper. Geoinformática. Planejamento de viagens.

Abstract

This work presents the development of an extensible system to support geospatial decision-making in the context of travel planning. The Terraplanner platform, originally unextensible, was adapted to support dynamic loading of frontend plugins. As a proof of concept, the TerraTripper plugin was implemented, allowing users to create personalized travel itineraries based on parameters such as time, cost, desired attractions, and accommodation points. The developed software architecture promoted modularity, component reuse, and seamless integration with the main system. The development process included the creation of a Requirements Specification Document (RSD), the implementation of a functional MVP, and its technical validation. The results demonstrated the feasibility of the proposed approach and its potential for expansion to other domains of geospatial decision support.

Keywords: SDSS. Plugins. Terraplanner. TerraTripper. Geoinformatics. Travel Planning.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

A tomada de decisão espacial sempre representou um desafio recorrente em diversas áreas do conhecimento, como planejamento urbano, logística, meio ambiente, saúde pública e turismo. Com o avanço das tecnologias de informação e comunicação, surgiram ferramentas capazes de integrar dados geoespaciais a processos decisórios. No entanto, observou-se, ao longo do tempo, a carência de soluções computacionais que oferecessem flexibilidade arquitetural, modularidade e facilidade de extensão para diferentes domínios de aplicação. Iniciativas como os Sistemas de Informação Geográfica (SIGs) e os Sistemas de Apoio à Decisão Espacial (SDSS) consolidaram-se como pilares na análise e visualização de informações georreferenciadas (JANKOWSKI et al., 1997).

Entretanto, muitos desses sistemas apresentaram barreiras de acesso técnico, arquiteturas pouco extensíveis e dificuldade de adaptação a problemas específicos. Identificou-se, portanto, uma demanda crescente por frameworks que não apenas integrassem dados espaciais, mas que também permitissem o desenvolvimento de aplicações customizadas por meio de componentes reutilizáveis e independentes.

Diante desse cenário, este trabalho tem como proposta o desenvolvimento de um framework extensível para apoio à tomada de decisão espacial, baseado em uma arquitetura de plugins dinâmicos, em que o front-end permite o carregamento de extensões modulares registradas por meio de uma API de gerenciamento de extensões. Essa abordagem tem como objetivo possibilitar a criação de aplicações específicas sobre uma base comum de serviços geoespaciais na WEB, promovendo reuso, modularidade e adaptação para diferentes domínios.

A arquitetura de software implementada neste trabalho permite que usuários cadastrem novos plugins via interface gráfica, os quais são armazenados e versionados por um backend dedicado, viabilizando a composição dinâmica de funcionalidades geográficas. Um aspecto fundamental desta arquitetura está na forma como o sistema gerencia as dependências externas dos plugins. Para garantir compatibilidade e funcionamento correto, o front-end expôs um conjunto de bibliotecas essenciais, como React (React Team, 2026), ReactDOM e outros componentes compartilhados, que foram injetados no ambiente global e disponibilizados automaticamente aos plugins carregados. Desta maneira, passou a ser possível desenvolver plugins em TypeScript/React de forma desacoplada e independente, desde que suas dependências estivessem previamente compartilhadas pela aplicação principal. Essa estratégia evitou redundância de código, conflitos de versão e contribuiu para uma execução mais estável e leve dos módulos externos.

Para demonstrar a viabilidade desta arquitetura, foi implementado um plugin denominado TerraTripper, voltado ao planejamento de viagens por meios de transportes terrestres. Esse plugin utilizou o frontend React e a infraestrutura de serviços do sistema Terraplanner, desenvolvido

pelo laboratório Terralab, colocando à prova a extensibilidade da arquitetura e demonstrando seu uso para o desenvolvimento de sistemas de apoio à tomada de decisões espaciais. Por meio desta aplicação, foi possível validar a abordagem de injeção de dependências, comunicação com geoserviços web e composição de funcionalidades geoespaciais de forma modular e incremental.

Dada a necessidade de especializar ferramentas de apoio à decisão espacial para diferentes domínios, sem aumentar o acoplamento e o custo de manutenção do núcleo, este trabalho investiga: como viabilizar, em um SDSS Web, a personalização por meio de uma arquitetura extensível baseada em plugins, atendendo simultaneamente (i) usuários finais, que executam decisões espaciais orientadas ao domínio, e (ii) desenvolvedores/empresas de TI, que implementam e distribuem extensões com integração controlada aos serviços do sistema?. Ao longo deste trabalho, investiga-se a hipótese de que uma arquitetura de microkernel, com contratos explícitos e injeção controlada de dependências no frontend, permite a evolução independente de plugins especializados por domínios preservando a estabilidade do núcleo, dando origem a um SDSS extensível e flexível no qual adições ou evoluções de funcionalidades não implicarão em re-codificações que se propagam em avalanches.

1.1 Justificativa

A crescente demanda por soluções geoespaciais específicas e a diversidade dos problemas que envolvem análise espacial evidenciaram a necessidade de arquiteturas que suportassem personalização e extensão. Ferramentas genéricas, como Google Maps ou MyMaps, embora amplamente utilizadas, não ofereceram suporte nativo à construção colaborativa de critérios personalizados de decisão nem permitem a extensão de suas funcionalidades via componentes externos. A ausência de frameworks flexíveis nesse contexto dificulta a criação de soluções adaptadas a diferentes usuários, restringindo o potencial de inovação em áreas que demandam análise espacial detalhada. Ao propor uma arquitetura extensível com base em plugins, este trabalho busca preencher essa lacuna, oferecendo uma fundação reutilizável sobre a qual diferentes aplicações geográficas podem ser construídas.

1.2 Objetivos

Esta seção apresenta o objetivo geral deste trabalho e também seus objetivos específicos.

1.2.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver um framework extensível para apoio à tomada de decisão espacial, utilizando arquitetura de plugins no front-end e integrando serviços WEB geoespaciais via backend. Então, avaliá-lo por meio do desenvolvimento de plugin voltado ao planejamento de viagens terrestres, um caso de muito comum e relevante de tomada de decisão

presente em diferentes domínios da atividade humana, incluindo lazer, roteamento de frotas e mobilidade urbana.

1.2.2 Objetivos Específicos

Os itens a seguir são os objetivos específicos deste trabalho:

- Definir requisitos de extensibilidade orientados a dois perfis de usuário: (i) usuário final do SDSS Web (apoio à decisão no domínio) e (ii) desenvolvedor/empresa de TI (criação, evolução e distribuição de extensões);
- Projetar e implementar uma arquitetura extensível baseada em plugins para o front-end de um conjunto de serviços Web geoespaciais, permitindo especialização por domínio sem modificações no núcleo do sistema;
- Especificar contratos de integração (APIs, hooks e componentes) e mecanismos de comunicação entre plugins e os serviços base do host;
- Definir e implementar mecanismos de governança técnica, incluindo injeção controlada de dependências, compatibilidade e integração segura com bibliotecas e serviços compartilhados;
- Criar uma API Web dedicada ao gerenciamento, armazenamento e versionamento de plugins, viabilizando seu carregamento dinâmico em tempo de execução;
- Implementar uma base de serviços reutilizáveis voltados à análise espacial para suportar o reuso por múltiplas aplicações e domínios;
- Desenvolver um plugin para planejamento de viagens terrestres como prova de conceito da arquitetura proposta, avaliando simultaneamente (i) a utilidade para o usuário final e (ii) a viabilidade/produtividade para o desenvolvedor;
- Avaliar vantagens, limitações e flexibilidade da arquitetura proposta, incluindo análise de trade-offs em comparação com abordagens alternativas de customização;
- Documentar serviços, contratos e interfaces para reuso por futuras aplicações e para manutenção e evolução do ecossistema de plugins.

1.3 Estrutura Do Trabalho

O Capítulo 2 apresenta a revisão bibliográfica com ênfase em SIG, SDSS, MCDA, arquiteturas plugáveis e planejamento de rotas. O Capítulo 3 descreve o desenvolvimento: os geo-serviços TerraPlanner com suporte a plugins, concepção do TerraTripper, fluxo de uso e

metodologia de desenvolvimento. O Capítulo 4 traz os resultados e a avaliação técnica destes resultados. Por fim, o último capítulo apresenta as considerações finais e as propostas de trabalhos futuros.

2 Revisão Bibliográfica

Este capítulo apresenta os conceitos fundamentais necessários ao entendimento do trabalho e discute trabalhos correlatos. Ele fornece o embasamento teórico para as decisões de projeto do framework proposto e do plugin TerraTripper, com ênfase em: (i) sistemas de apoio à decisão espacial, (ii) arquiteturas de software extensíveis baseadas em plugins e (iii) estratégias de integração de componentes geoespaciais em aplicações web. Além disso, destaca-se requisitos não funcionais recorrentes em ecossistemas plugáveis — como versionamento, compatibilidade, isolamento e governança — critérios estes que orientaram a implementação descrita no Capítulo 3.

2.1 Conceitos Básicos

Esta seção de texto apresenta e descreve os conceitos fundamentais necessários ao embasamento deste trabalho, apontando as referências utilizadas durante o estudo.

2.1.1 Sistemas de apoio à decisão (DSS) e apoio à decisão espacial (SDSS)

Os Sistemas de Apoio à Decisão (Decision Support Systems – DSS) foram discutidos como uma evolução dos sistemas de informação gerenciais, com ênfase em apoiar decisões semi-estruturadas por meio de dados, modelos e forte interação com o tomador de decisão (GORRY; MORTON, 1971; SPRAGUE, 1980). Nessa linha, a literatura ressaltou que o valor de um DSS não esteve apenas em produzir uma "resposta ótima", mas em permitir exploração, comparação de alternativas e aprendizado ao longo do processo decisório.

Quando a dimensão geográfica foi incorporada explicitamente, a pesquisa em Spatial Decision Support Systems (SDSS) descreveu requisitos adicionais, como representação de fenômenos espaciais, integração com operações de SIG, múltiplos critérios e, frequentemente, múltiplos atores e preferências (DENSHAM; GOODCHILD, 1989; ARMSTRONG, 1994). Em particular, a visualização cartográfica e a interação com mapas foram apontadas como elementos essenciais para apoiar análise exploratória e comunicação de resultados em problemas espaciais.

No contexto deste trabalho, o referencial de SDSS foi utilizado para orientar requisitos do framework: a solução deveria suportar interação espacial no navegador, permitir a incorporação incremental de métodos/visões analíticas e, ao mesmo tempo, manter uma base arquitetural estável para evoluções futuras.

2.1.2 Sistemas de Informação Geográfica (SIG) e informação geoespacial

Os Sistemas de Informação Geográfica (SIG) foram caracterizados como conjuntos de métodos e ferramentas para captura, armazenamento, consulta, análise e visualização de dados georreferenciados, sustentando operações como sobreposição de camadas, análises de proximidade e representação cartográfica (LONGLEY et al., 2015). A literatura também destacou que informação geoespacial envolve modelos de dados (vetorial e matricial), diversos sistemas de referência cartográfica, metadados e incertezas, o que impacta diretamente o modo como análises e decisões são produzidas e interpretadas.

Para o framework desenvolvido, esses conceitos foram relevantes ao orientar a separação de responsabilidades: funcionalidades transversais (por exemplo, visualização de camadas e interação com mapa) foram tratadas como parte do núcleo, enquanto regras e fluxos de decisão específicos de um domínio puderam ser implementados como plugins.

2.1.3 MCDA em SIG e suporte multicritério

Na literatura, a análise de decisão multicritério (Multi-Criteria Decision Analysis – MCDA) aplicada a SIG foi amplamente empregada para estruturar problemas em critérios, ponderações e regras de combinação, permitindo comparar alternativas sob múltiplas dimensões (MALCZEWSKI, 1999). Revisões indicaram que a integração GIS–MCDA consolidou-se como base para apoiar planejamento territorial e seleção de locais, combinando técnicas como ponderação linear, AHP e métodos outranking com operações espaciais e visualização (MALCZEWSKI, 2006).

Embora o presente trabalho não tenha implementado um módulo completo de MCDA, a revisão foi útil para definir requisitos de extensibilidade: o framework deveria permitir que plugins adicionassem critérios, dados complementares, indicadores e métodos de agregação sem exigir mudanças no núcleo, evitando acoplamentos que dificultassem evolução incremental.

2.1.4 WebGIS e interoperabilidade por padrões

A evolução para WebGIS deslocou funcionalidades de SIG para ambientes distribuídos, habilitando acesso por navegadores e integração de serviços geoespaciais. Trabalhos sobre Internet GIS discutiram o papel de arquiteturas distribuídas e serviços para disponibilização e consumo de informação geográfica em rede (PENG; TSOU, 2003). Nesse contexto, padrões do Open Geospatial Consortium (OGC), como WMS (Web Map Service) e WFS (Web Feature Service), foram amplamente adotados para favorecer interoperabilidade e reutilização de dados e serviços entre diferentes sistemas (Open Geospatial Consortium, 2006; Open Geospatial Consortium, 2010).

Para o framework proposto, essa base sustentou a decisão de isolar contratos de acesso a dados e serviços (back-end/API) de detalhes internos do front-end, possibilitando que plugins

consumissem recursos por interfaces de software estáveis. Ficou claro que o versionamento de interface seria necessário em cenários de evolução dos provedores de dados/serviços de software que seria integrados aos plugins desenvolvidos pelo usuário .

2.1.5 Participação, VGI e colaboração

A discussão sobre participação e produção colaborativa de dados reforçou que, em muitos domínios, usuários passaram a atuar como produtores e validadores de informação geográfica. (GOODCHILD, 2007) descreveu o fenômeno de Volunteered Geographic Information (VGI) e suas implicações para qualidade, atualização e participação. Esse cenário reforçou a necessidade de ferramentas flexíveis, capazes de incorporar rapidamente novas funcionalidades e fluxos de interação, sobretudo quando o domínio muda com frequência ou quando novos perfis de usuário são incorporados.

2.1.6 Arquiteturas extensíveis, frameworks e plugins

Do ponto de vista da Engenharia de Software, a extensibilidade foi analisada a partir de princípios de modularidade e encapsulamento. O critério de decomposição por ocultação de informação (information hiding) defendeu que os módulos de software deveriam esconder decisões de projeto propensas àmudanças, expondo apenas interfaces estáveis (PARNAS, 1972)). A literatura de padrões e arquitetura de projetos de software também descreveu estruturas capazes de suportar evolução incremental, como arquiteturas em camadas e o padrão microkernel, no qual um núcleo mínimo fornece serviços comuns e extensões adicionam funcionalidades específicas ao redor desse núcleo (BUSCHMANN et al., 1996).

Frameworks foram descritos como estruturas reutilizáveis que definem pontos de extensão (hotspots) e fornecem serviços comuns, ao passo que plugins implementam variações e funcionalidades específicas sem exigir alteração do núcleo (GAMMA et al., 1994; SZYPERSKI, 2002). Na prática, ecossistemas como o Eclipse popularizaram plataformas baseadas em plugins, com carregamento dinâmico e contratos explícitos para extensões, facilitando manutenção e evolução do sistema ao longo do tempo (MCAFFER; LEMIEUX, 2005).

No contexto deste trabalho, esses conceitos foram aplicados para definir (i) um núcleo estável responsável por infraestrutura de navegação, layout e serviços compartilhados e (ii) um mecanismo de carregamento dinâmico de plugins no frontend, no qual cada plugin forneceu sua própria interface e se integrou ao host por meio de dependências expostas e contratos explícitos (interfaces), reduzindo o acoplamento direto entre plugin e núcleo.

2.1.7 Síntese conceitual aplicada ao problema

A partir dessa revisão, o desenvolvimento do framework foi orientado por duas decisões centrais: (1) manter o núcleo o mais estável e pequeno possível, concentrando nele apenas

serviços transversais (por exemplo, autenticação, roteamento, provedores de estado e integração de mapa) e (2) tornar explícitos os pontos de extensão do sistema por meio de interfaces e contratos de dependências, para que novas funcionalidades fossem adicionadas como plugins, sem necessidade de modificar o host. Esse desenho foi coerente com o princípio de ocultação de informação (PARNAS, 1972) e com abordagens de microkernel e frameworks extensíveis (BUSCHMANN et al., 1996).

Embora a literatura apresente a arquitetura plugável como alternativa direta à complexidade da evolução de sistemas monolíticos, a adoção desse estilo impõe desafios adicionais. Em particular, ambientes com carregamento dinâmico demandam mecanismos explícitos de compatibilidade entre versões, gestão de dependências compartilhadas, tratamento de falhas (isolamento de exceções) e observabilidade do ciclo de vida dos plugins. Esses aspectos tornam-se especialmente relevantes quando o objetivo é permitir evolução incremental do sistema sem interromper o uso por diferentes perfis de usuários.

2.2 Trabalhos Correlatos

Os trabalhos correlatos foram analisados a partir de três eixos: (i) ambiente de execução (desktop ou web), (ii) mecanismo de extensibilidade (plugins, módulos ou composição por serviços) e (iii) adequação ao contexto de apoio à decisão espacial, incluindo integração com dados, serviços e padrões geoespaciais. O objetivo foi situar o framework proposto em relação a plataformas consolidadas e identificar lacunas que motivaram a abordagem adotada.

Uma referência importante de extensibilidade em SIG foi o projeto QGIS, que apresentou um ecossistema maduro de plugins e uma API consolidada para extensão e automação (QGIS Development Team, 2026). O QGIS favoreceu a rápida ampliação de funcionalidades, porém foi centrado em uma aplicação desktop; consequentemente, o carregamento remoto de extensões no cliente e o compartilhamento em ambiente web não foram o foco principal, o que diferiu do objetivo deste trabalho.

Outra linha de plataformas baseadas em plugins foi exemplificada por aplicações construídas sobre o Eclipse Rich Client Platform (RCP), que adotaram modularização por bundles e pontos de extensão. Esse modelo reforçou a viabilidade de um núcleo mínimo estendido por componentes e destacou benefícios de contratos explícitos e versionamento de extensões (MCAFFER; LEMIEUX, 2005). Entretanto, o ciclo de distribuição típico permaneceu associado a aplicações instaláveis, enquanto o presente trabalho buscou a extensibilidade diretamente no frontend web.

No campo de plataformas WebGIS, projetos como GeoNode organizaram componentes para catálogo, publicação e visualização de camadas geoespaciais, priorizando colaboração e integração por serviços. Nesses casos, a extensibilidade foi frequentemente obtida pela composição de serviços e pela adição de módulos no backend, e não por um mecanismo de plugins de UI carregados dinamicamente no cliente (GeoNode Project, 2024).

Em síntese, os trabalhos correlatos evidenciaram dois pontos: (a) plataformas consolidadas ofereceram extensibilidade robusta, porém frequentemente em contexto desktop ou com extensão concentrada no backend; e (b) soluções WebGIS priorizaram integração e publicação de dados, mas nem sempre forneceram um modelo explícito de plugins no frontend para evolução incremental e isolada de funcionalidades. O framework proposto buscou contribuir nesse espaço ao tratar a interface web como ponto central de extensibilidade, preservando um núcleo estável e contratos bem definidos para a integração de plugins.

Nos frameworks de plugins frontend, a extensibilidade ocorre no mesmo ambiente de execução do navegador, o que amplia a superfície de risco e exige cuidados de segurança. Entre estratégias frequentes estão: validação e assinatura de artefatos, limitação de permissões expostas ao plugin (princípio do menor privilégio), sandboxing por meio de iframes quando apropriado e definição de contratos estáveis para comunicação entre host e plugin (OWASP, 2026). No contexto deste trabalho, tais preocupações foram tratadas principalmente pela restrição de dependências que seriam expostas para os plugins e pela centralização do carregamento e do registro de plugins no núcleo da arquitetura de extensibilidade, conforme detalhado no Capítulo 3.

2.3 Considerações

A análise de distintos campos da ciência e tecnologias fundamentou o projeto do framework extensível para tomada de decisão espacial desenvolvido neste trabalho. Esta análise orientou a definição de contratos e de integração entre o núcleo da arquitetura para extensibilidade e os plugins, bem como orientou a escolha por uma abordagem frontend para permitir a expansão de funcionalidades sem acoplamento rígido aos serviços espaciais (backends). A revisão sobre SDSS, SIG e WebGIS, MCDA e VGI permitiu a elicitação de requisitos de interação espacial, de interoperabilidade e de apoio à decisão. No capítulo seguinte, essas premissas são retomadas na descrição detalhada da arquitetura implementada no produto TerraPlanner e também do plugin TerraTripper desenvolvido como prova de conceito desta arquitetura.

3 Desenvolvimento do Framework Extensível TerraPlanner/TerraTripper

Este capítulo descreve detalhadamente o projeto da arquitetura de extensibilidade implementada neste trabalho, assim como descreve o plugin desenvolvido para para avaliar o uso desta arquitetura no desenvolvimento de sistemas de suporte à decisão espacial. Ao longo do capítulo, as escolhas de tecnologias e decisões de projetos são justificadas.

3.1 O framework TerraPlanner para Planejamento espacial

Este trabalho desenvolveu uma arquitetura plugável que estendeu o produto TerraPlanner (www.terraplanner.org), desenvolvido pelo laboratório TerraLAB, do Departamento de Computação, da Universidade Federal de Ouro Preto. Este produto oferece um grande conjunto de funcionalidades que justificam sua utilização para o desenvolvimento de um SDSS como o que este trabalho pretende.

O sistema apresenta um frontend web rico que permite a interação espacial do usuários com mapas dinâmicos renderizados em tempo real, oferece interoperabilidade com os principais formatos de dados espaciais definidos pela OGC e se integra a outros serviços espaciais via protocolos também definidos pela OGC, como o WMS. O frontend é moderno, desenvolvido na linguagem TypeScript através do uso do framework REACT (React Team, 2026). Esse frontend também se integra a diversos serviços web para aquisição, armazenamento, análise e visualização de dados espaciais, além de serviços para agregação e sumarização de dados socioeconômicos (áreas de influência), serviços para planejamento de rotas e serviços para computação das regiões alcançadas (ou simplesmente, alcances) por diferentes tipos de veículos a partir de um epicentro, considerando a malha de estradas existente na região. No entanto, este produto não dispõe de formas nativas de estender suas funcionalidades ou de especializá-las para um domínio a fim de gerar produtividade.

Durante o desenvolvimento da arquitetura para extensibilidade do TerraPlanner, um dos principais objetivos foi desacoplar as funcionalidades principais de navegação pelo mapa — manipulação de camadas de dados, planejamento de rotas, computação de alcances, análise de de dados socioeconômicos regionais (áreas de influência) — da lógica específica implementada de cada funcionalidade, além de permitir que novos módulos sejam adicionados sem alterações no núcleo da arquitetura.

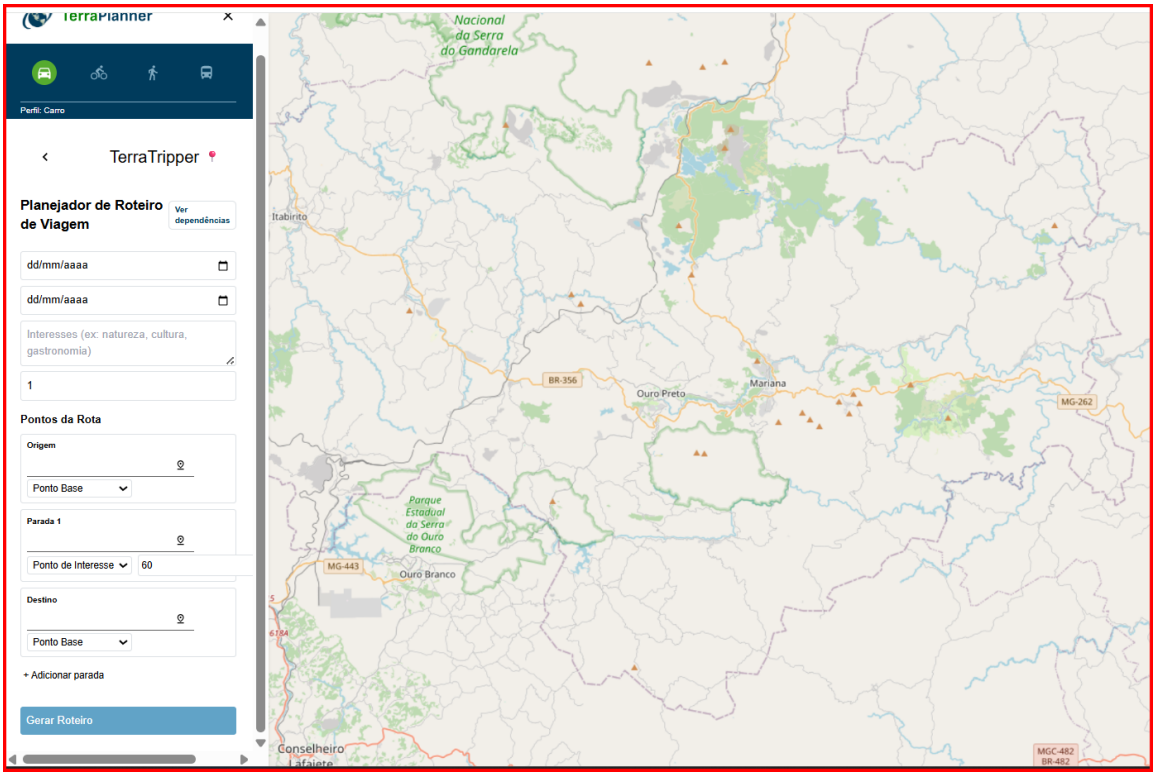

Figura 3.1 – Tela do TerraPlanner com o plugin TerraTripper em uso (área de plugin à esquerda e mapa base à direita).

3.2 Visão geral da arquitetura plugável

A Figura 3.2 apresenta a visão geral da arquitetura plugável implementada. O diagrama organiza o fluxo de dados e as responsabilidades dos componentes envolvidos no cadastro, entrega e execução de plugins em tempo de execução no TerraPlanner.

De forma resumida, a arquitetura foi estruturada em três partes: (i) o TerraPlanner Host (frontend), que carrega e executa plugins; (ii) um serviço WEB (backend) dedicado ao gerenciamento e armazenamento de plugins chamado Plugin API/Store); e (iii) os artefatos de plugins, armazenados e servidos sob demanda.

O fluxo principal ocorre em duas operações complementares. Primeiro, no cadastro de plugins, o usuário fez upload de arquivos .ts/.tsx (um único arquivo ou um conjunto contendo componentes auxiliares) por meio de um componente de upload no frontend. O TerraPlanner encaminha esses arquivos para a Plugin API/Store, que persiste seu conteúdo e registra o plugin na arquitetura. Depois, na proxima vez que o frontend for iniciado, durante seu carregamento, o TerraPlanner consulta a Plugin API para obter a lista de plugins registrados e, com base nessa lista, atualiza seu próprio menu dinamicamente, ofertando cada plugin como uma opção do menu. Quando o usuário seleciona um determinado plugin, o frontend requisita o ponto de entrada do plugin (por exemplo, GET /plugins/{id}/main.js) e o executa no navegador.

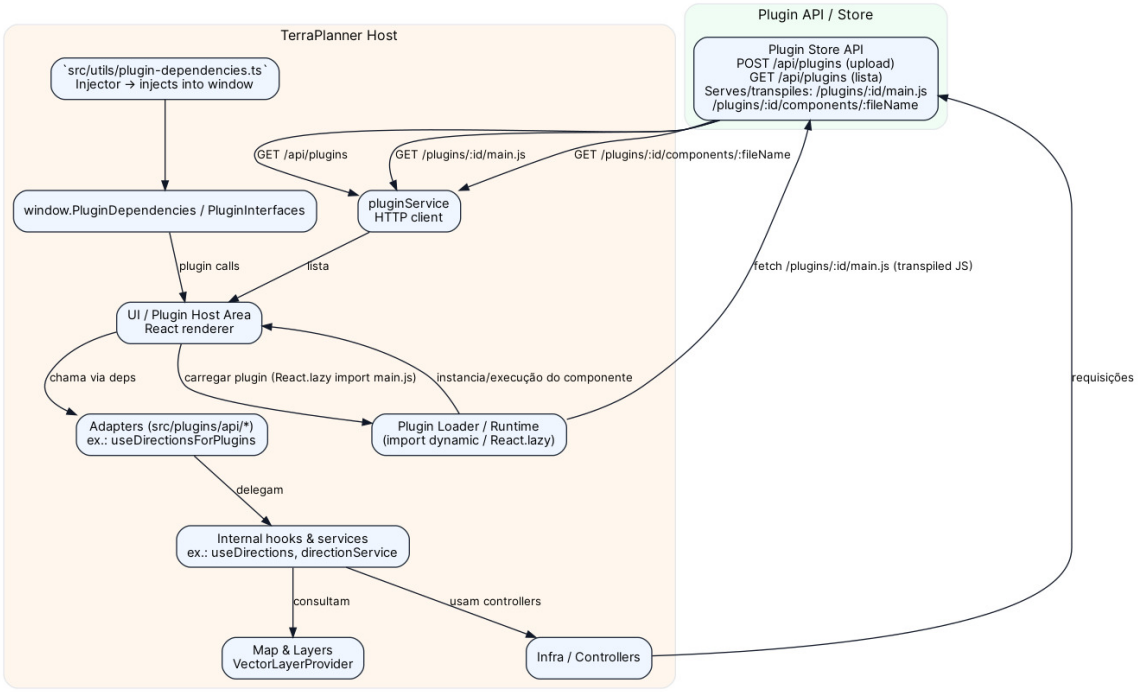

Figura 3.2 – Diagrama fluxo de dados da arquitetura de plugins TerraPlanner

3.3 Separação por camadas e responsabilidades

A arquitetura é organizada por camadas para reduzir acoplamentos entre o núcleo do TerraPlanner e os módulos adicionados. O host permanece responsável por orquestrar o carregamento, oferecer a infraestrutura de interface e integrar serviços espaciais. Já o plugin permanece responsável por implementar fluxos e telas orientados ao domínio, reutilizando as capacidades do host por meio de interfaces estáveis.

A Figura 3.3 apresenta a separação por camadas e evidencia o papel de contratos e adaptadores como fronteira entre host e plugins. Essa fronteira permite que alterações internas do TerraPlanner sejam absorvidas por adaptadores, preservando a estabilidade dos pontos de extensão consumidos pelos plugins.

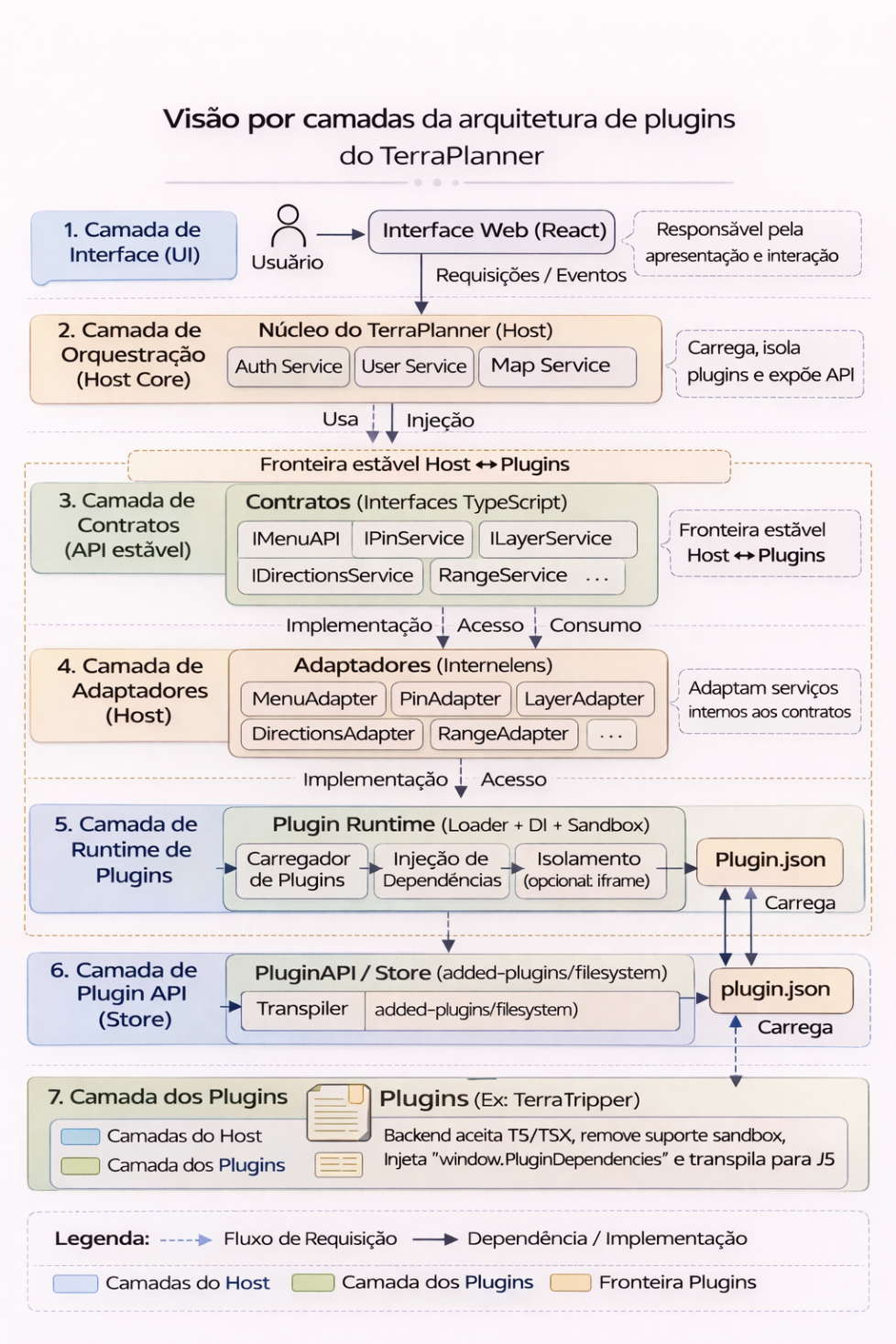

Figura 3.3 – Visão por camadas: UI, host, runtime de plugins, contratos/adaptadores e serviços do núcleo.

3.4 Contratos, adaptadores e injeção de dependências

Para limitar o acoplamento e tornar explícitas as capacidades do TerraPlanner disponíveis ao plugin, foram definidos contratos na forma de interfaces e hooks que o frontend expõe ao ambiente de execução dos plugins. Esses contratos descrevem operações relacionadas à

manipulação do menu, à manipulação do mapa, à manipulação de pins, ao controle de camadas de dados geográficos, ao serviço de planejamento de rotas e às funcionalidades de computação de alcances. No host, adaptadores conectam esses contratos às implementações desenvolvidas pela equipe de frontend do TerraPlanner, sem que o código legado precise ser alterado.

Como estratégia para garantir compatibilidade e para evitar conflitos de dependências, o host expõe um conjunto predefinido e versionado de serviços, hooks e componentes por meio do objeto global window.PluginDependencies. Assim, o plugin consume funcionalidades do TerraPlanner apenas por contratos predefinidos, reduzindo a necessidade de empacotar dependências duplicadas e preservando a governança do núcleo sobre o que é acessível em tempo de execução. A Tabela 3.1 enumera as operações mais relevantes de cada contrato do frontend TerraPlanner que este trabalho expõe. O diagrama de classes da Figura 3.4 apresenta a interface de programação completa dos contratos desenvolvidos.

Tabela 3.1 – Contratos expostos pelo TerraPlanner e operações relevantes

Contrato	**Responsabilidade principal**	**Operações relevantes**
`IMenuAPI`	Controle do menu e navegação do TerraPlanner	• `selectedMenuOption` • `changeMenuOption(option)`
`IUsePinHook`	Manipulação de pins no mapa	• `getPinById(id)` • `addPin(id, params)` • `removePin(id)` • `updatePin(id, params)`
`IUseLayerHook`	Controle de visibilidade de camadas de dados geográficos	• `getSource()` • `setVisible(boolean)` • `getVisible()`
`IUseDirectionsHook`	Cálculo e renderização de rotas	• `fetchDirections(params)` • `renderDirections(data)` • `clearDirections()`
`IUseRangeHook`	Cálculo e exibição de alcances	• `fetchRange(params)` • `renderRanges(data)` • `clearSource()`

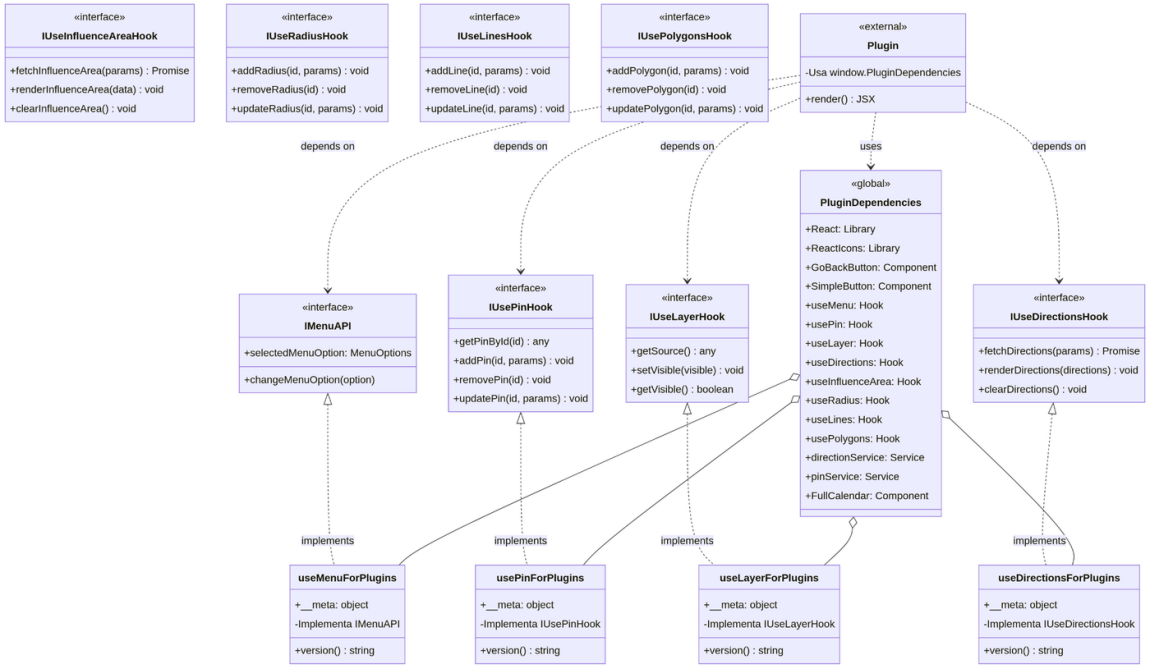

Figura 3.4 – Diagrama de contratos oferecidos aos plugins

3.5 Serviço para gestão de plugins (Plugin API/Store)

A arquitetura possui seu próprio backend que executa como um serviço WEB independente do backend geral do TerraPlanner, ele foi implementado um microsserviço dedicado à gestão de plugins denominado Plugin API/Store. Esse serviço recebe uploads do código dos plugins, persiste arquivos em disco e serve os artefatos dos plugin quando requisitados pelo frontend. Ao isolar a responsabilidade de extensibilidade em uma microsserviço específico, o backend do TerraPlanner não precisou incorporar detalhes operacionais sobre plugins, concentrando em um único componente as tarefas de registro e entrega de artefatos estendidos.

A Figura 3.5 mostra a interface gráfica com o usuário para upload e registro de plugins desenvolvida neste trabalho. Por meio dela, o programador fornece metadados do plugin que ele desenvolveu (atualmente, somente o nome do plugin), designa o ponto de entrada do plugin (isto é, o arquivo contendo seu código principal, por exemplo, “main.tsx”) e envia para registros todos os componentes (demais arquivos de código) que formam seu plugin. Uma vez registrado, o plugin passa a compor a lista retornada ao TerraPlanner Host, viabilizando a atualização dinâmica do menu. O diagrama de sequência apresentado na Figura 3.6 mostra as comunicações que ocorrem entre os componentes da arquitetura quando o upload e registro de um plugin é realizado pelo programador.

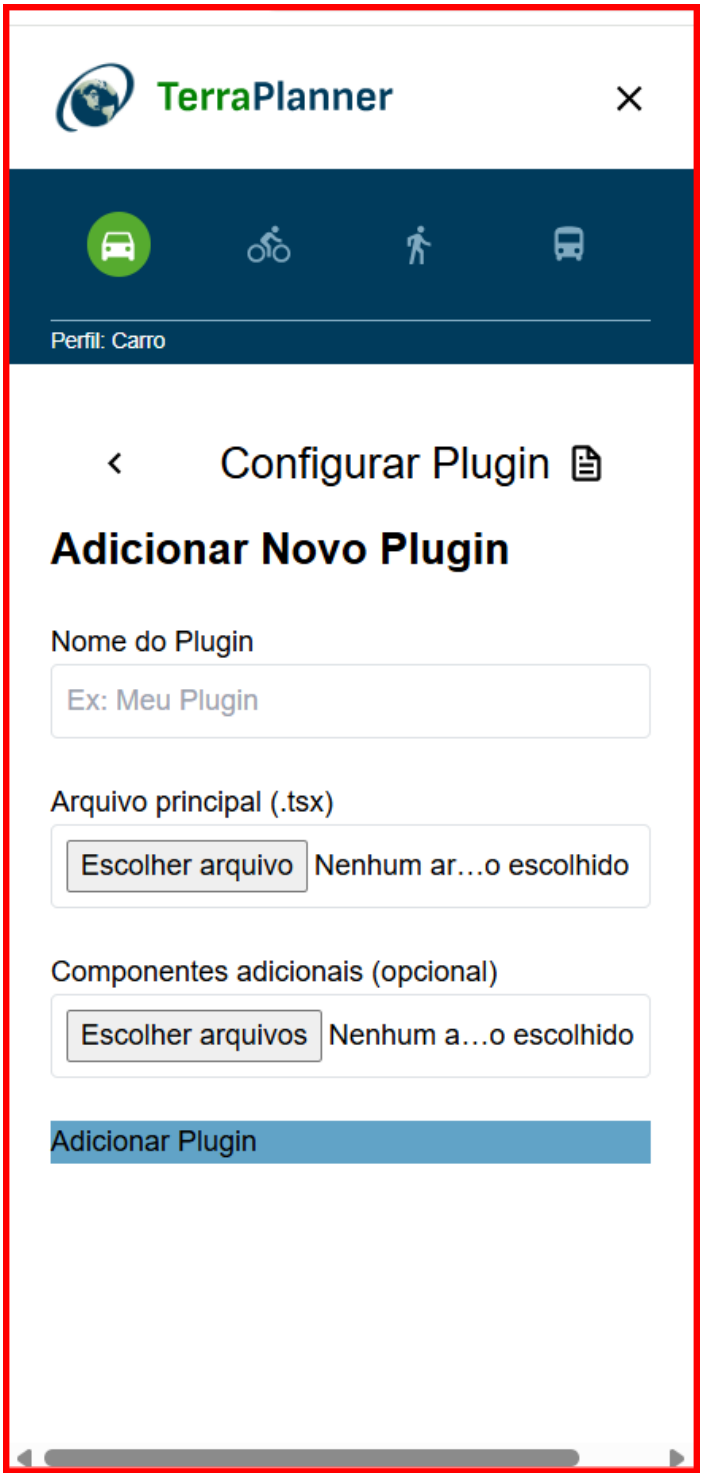

Figura 3.5 – Tela de upload e registro de plugins

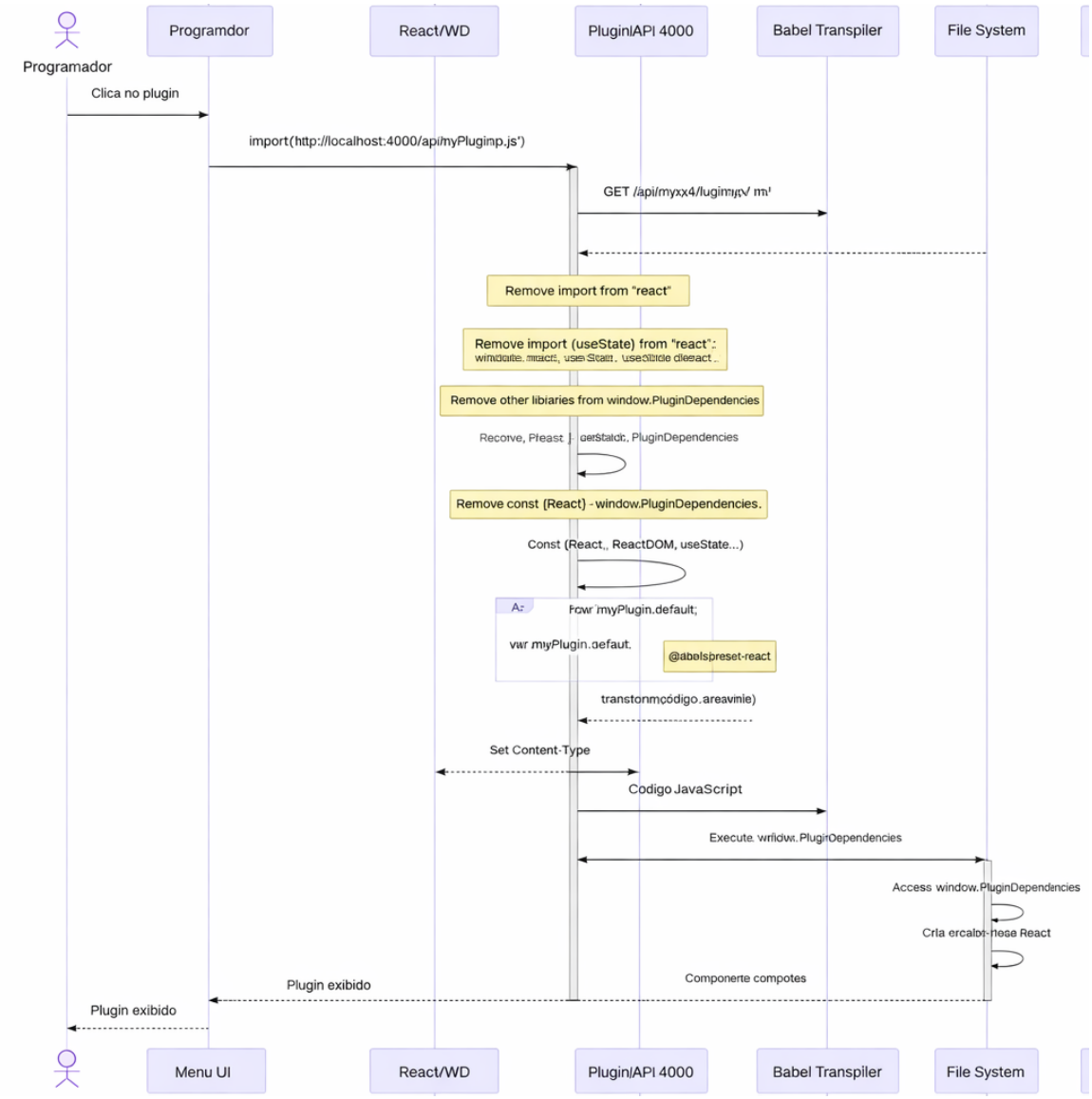

Figura 3.6 – Sequência de upload e registro de plugins

3.6 Carregamento dinâmico e transpilação de plugins em tempo de execução

O carregamento no frontend do TerraPlanner de um plugin registrado ocorre quando o usuário final seleciona esse plugin no menu. Neste momento, o frontend TerraPlanner aciona um carregador de plugins em tempo de execução que requisita à Plugin API/Store seu ponto de entrada. No backend, esse arquivo de código TypeScript/TSX é então transpilado para JavaScript compatível com o navegador, utilizando a dependência Babel para converter TS/TSX em JS e aplicar transformações necessárias para execução no runtime do host, como a injeção explícita de dependências (contratos) expostas pelo TerraPlanner.

Detalhadamente, durante o carregamento sob demanda de plugins, o componente Re-

act.lazy() emite uma requisição HTTP GET para /plugins/{id}/main.js, carregando seu ponto de entrada. No serviço Plugin API/Store, o servidor Express localiza o arquivo TSX correspondente, lê o conteúdo e aplica uma série de transformações: remoção de imports de dependências React e de bibliotecas já fornecidas pelo host, elimina declarações redundantes e realiza a injeção explícita de dependências através do objeto window.PluginDependencies da arquitetura. Então, um export default genérico é realizado e o código é transpilado com Babel (@babel/preset-react e @babel/preset-typescript), gerando JavaScript compatível com o navegador. O Serviço Plugin API/Store devolve o código JavaScript com o content-type adequado. No navegador, o módulo é avaliado pelo mecanismo de import dinâmico e o componente React exportado é instanciado dentro do contêiner de plugins. A partir desse momento, o plugin pode interagir com o Terra-Planner exclusivamente por meio dos contratos expostos em PluginDependencies, sem acesso direto às implementações internas.

A Figura 3.7 apresenta a sequência de comunicações de carregamento e evidencia o encadeamento de solicitações e transformações até a renderização do plugin no contêiner do host. O carregamento sob demanda foi acompanhado por decisões para também garantir desempenho e robustez: Após o primeiro carregamento, o plugin permanece em cache no ambiente do navegador, reduzindo a latência em acessos subsequentes. Essa abordagem também permie manter o bundle principal do TerraPlanner mais enxuto, enquanto habilita estratégias futuras de cache, a validação das versões de contratos demandados pelos plugins e monitoramento dos plugins registrados em tempo de execução.

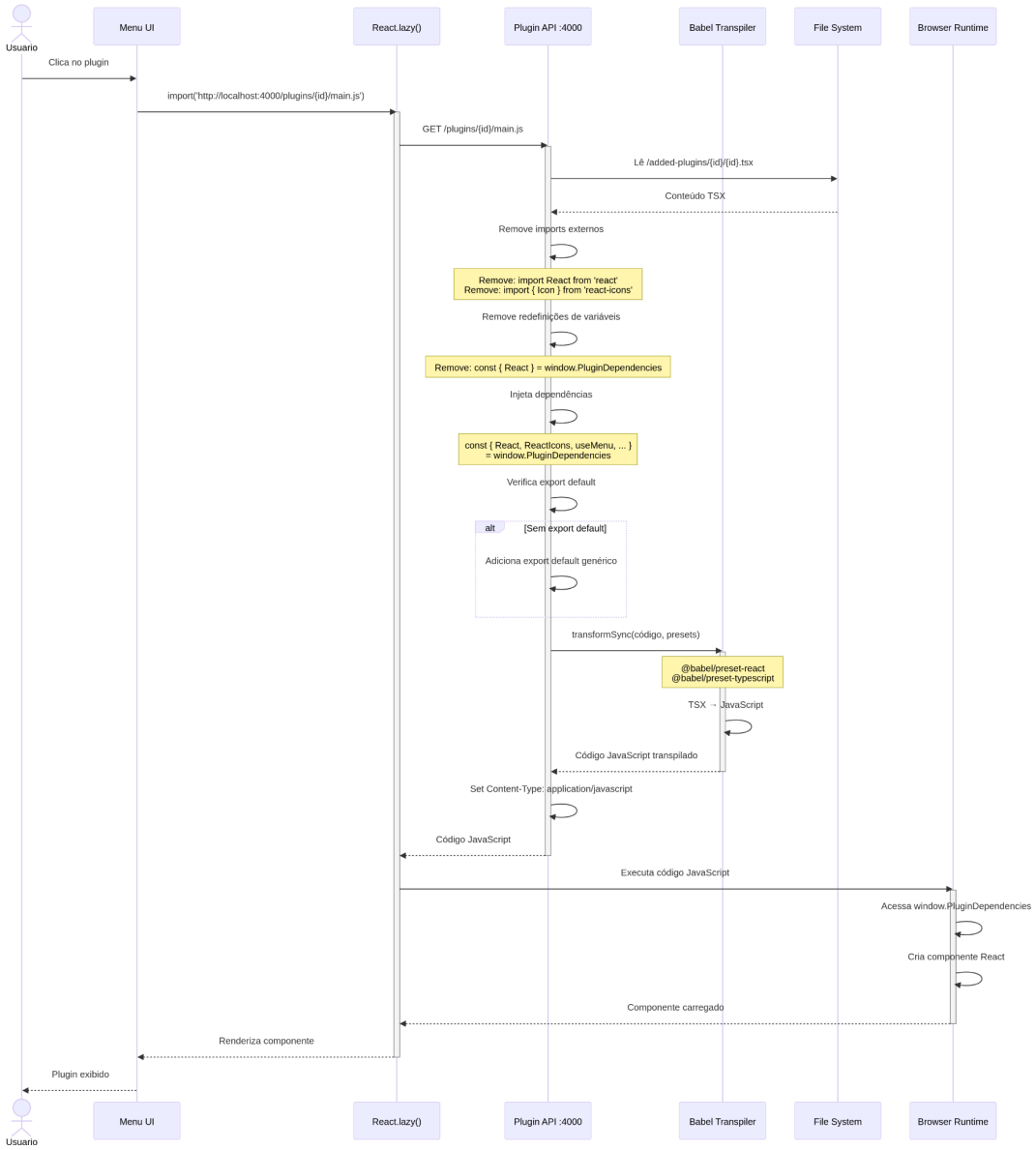

Figura 3.7 – Sequência de carregamento de um plugin React

3.7 Execução do plugin no host e integração com o mapa

Após carregado, o plugin se comporta como um componente React renderizado pelo próprio TerraPlanner, porém restrito às capacidades expostas por contratos. Desta maneira, as operações geoespaciais realizadas pelos plugins, como criação de pins ou solicitação de rotas, são na verdade executadas por meio dos serviços do host, e seus resultados são refletidos no mapa (OpenLayers) e nos demais elementos de interface gráfica do frontend TerraPlanner. A Figura 3.8 ilustra rotas de diferentes cores que o plugin TerraTripper solicitou que fossem renderizadas na área de mapa do frontend TerraPlanner. Ela também mostra que diferentes tipos de pins (circunferências e casinhas) foram criadas pelo plugin TerraTripper para demarcar pontos de interesse no mapa (atrativos turísticos e hospedagens, respectivamente). O diagrama de

sequência da Figura 3.9 mostra as comunicações que ocorrem quando um plugin utiliza contratos relacionados à manipulação de pins.

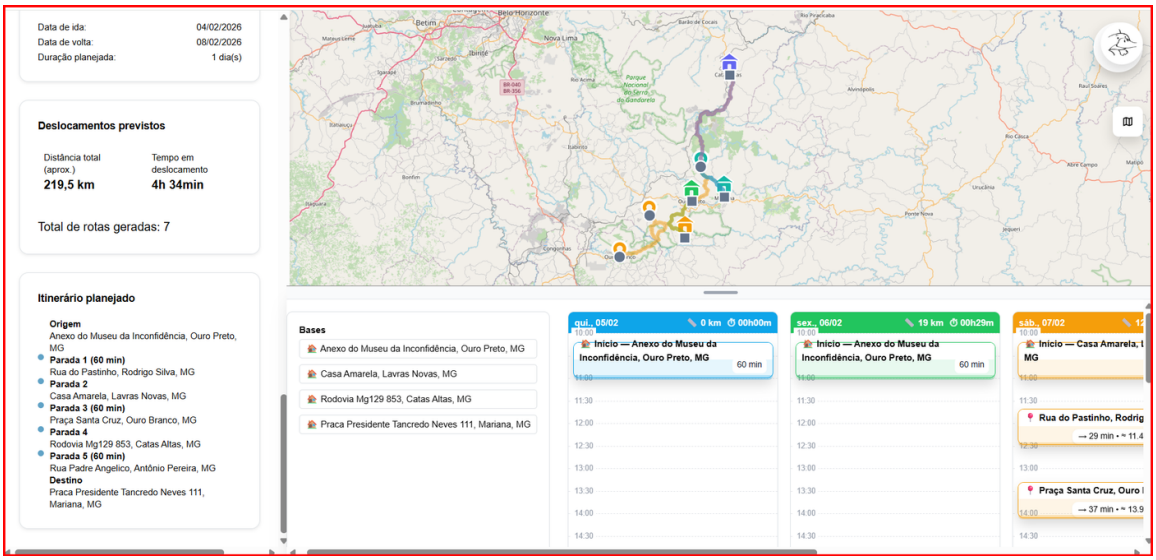

Figura 3.8 – Pins e rotas geradas pelo plugin TerraTripper na interface gráfica do frontend TerraPlanner.

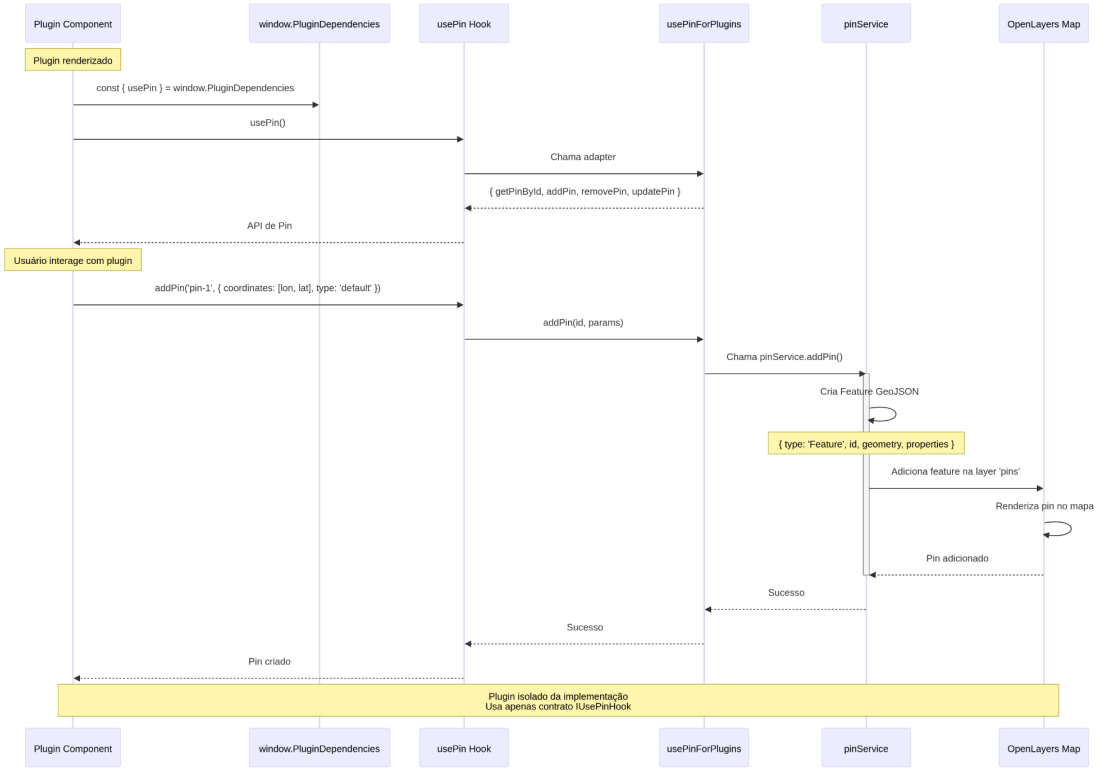

Figura 3.9 – Diagrama de sequência (UML) — fluxo de chamadas do contrato usePin para criação de pins no OpenLayers

3.8 Plugin TerraTripper como prova de conceito

Para avaliar a arquitetura plugável implementada, foi desenvolvido o plugin TerraTripper como prova de conceito. O TerraTripper implementa um planejador de viagens terrestres de múltiplos dias que considera que o usuário final visitará diversos locais, os dividindoem pontos de interesse (POIs como atrativos turísticos) e pontos base (por exemplo, hospedagens onde ele dormirá). Ele permite que o usuário estude diversas ordens de visita a estes locais, avaliando o tempo e distância necessária para realizar diferentes roteiros. Para isso, o TerraTripper combina em sua interface gráfica elementos para configuração de locais a serem visitados, um quadro de horários (calendário) para programação de locais a serem visitados por dia. A cada alteração de programação realizada pelo usuário, o TerraTripper utiliza a integração com serviços de rotas e de alcances expostos pelo host para recalcular o roteiro a ser seguido pelo usuário e o apresenta no mapa, junto com estimativas de tempo e distância. Desta maneira, o plugin TerraTripper busca evidenciar a capacidade da tecnologia produzida neste trabalho em subsidiar o desenvolvimento de SDSS .

A Figura 3.10 mostra o TerraTripper (à esquerda) em execução dentro do TerraPlanner, evidenciando a execução do plugin como parte da aplicação hospedeira e a reutilização das capacidades do host para visualização e interação no mapa.

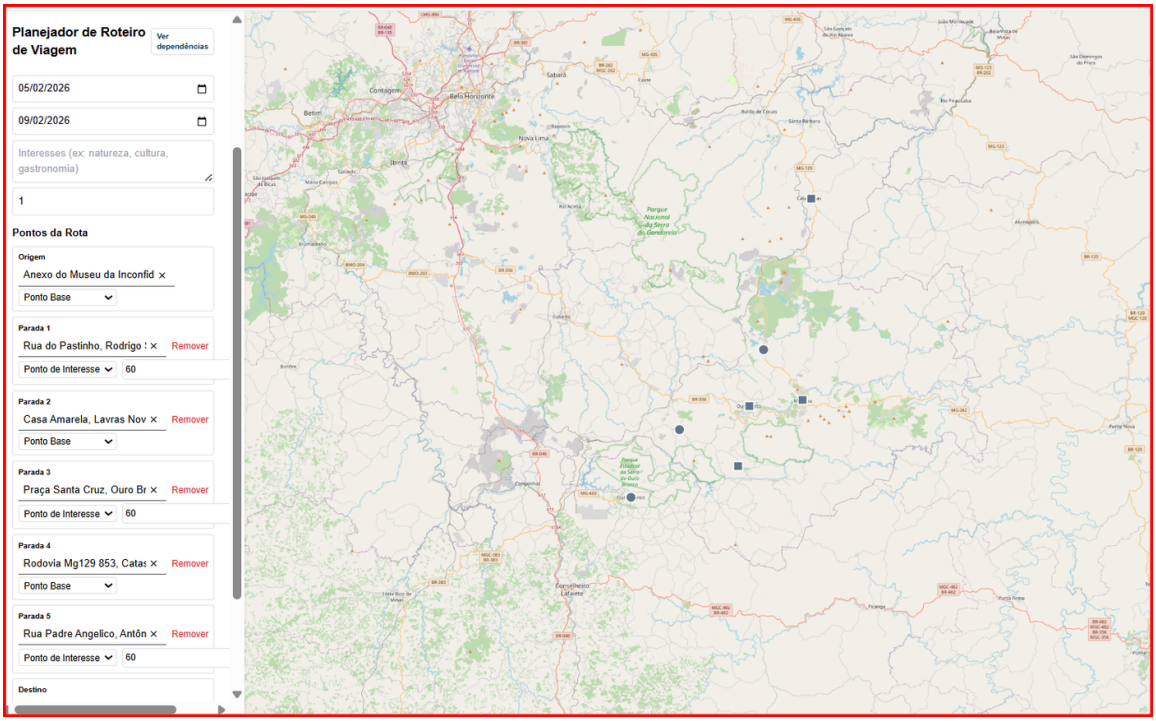

Figura 3.10 – O plugin TerraTripper em execução no TerraPlanner, com lista (a esquerda) e localização no mapa (à direita) de pontos de interesse (locais) a serem visitados e rotas exibidas no mapa.

3.8.1 Passo a passo do planejamento de viagens no TerraTripper

Para facilitar a compreensão do leitor, o fluxo de interações típico para uso do TerraTripper é descrito a seguir e pode ser acompanhado por capturas de tela (Figuras 3.11 a 3.15).

1. Configuração inicial (Figura 3.11): o usuário define origem, destino, período da viagem, veículo de deslocamento (carro, à pé, bicicleta ou ônibus) e elementos básicos de análise espacial (por exemplo, raio/réguas/alcances).

2. Seleção de bases e pontos de interesse (Figura 3.12): o usuário escolhe bases (fixas/reutilizáveis) e POIs, que são organizados em uma lista à esquerda da tela e também localizados no mapa.

3. Geração automática do roteiro (Figura 3.13): sob demanda, o plugin gera um roteiro inicial distribuindo POIs por dia, considerando restrições de base onde pernoites deverão acontecer e utilizando estimativas de deslocamento para computar rotas de forma minizar a distância global do roteiro. Este processamento acontece prioritariamente na máquina cliente, evitando sobrecarga dos backends do TerraPlanner.

4. Ajustes manuais no calendário (Figura 3.14): nem sempre o melhor roteiro é aquele que minimiza a distância ou tempo global da viagem. Na verdade, o melhor roteiro é algo muito subjetivo, alguns usuários irão preferir a menor distância, outros podem preferir a sequência de visitas mais divertida evitando viagens noturnas. Por isso, o TerraTripper permite que o usuário rearranje pontos base ou de interesse os arrastando (drag-and-drop) no calendário de um dia ou horário para outroO usuário também pode remover ou adicionar pontos a qualquer momento e pode ajustar tempos previstos de permanência em cada local.

5. Cálculo e visualização de indicadores para apoio a tomada de decisão (Figura 3.15): o plugin solicita rotas ao host a cada interação do usuário, então, as apresenta no mapa, além de apresentar estimativas de tempo e distância resumidas por dia e uma síntese gobal para todo oroteiro vigente.

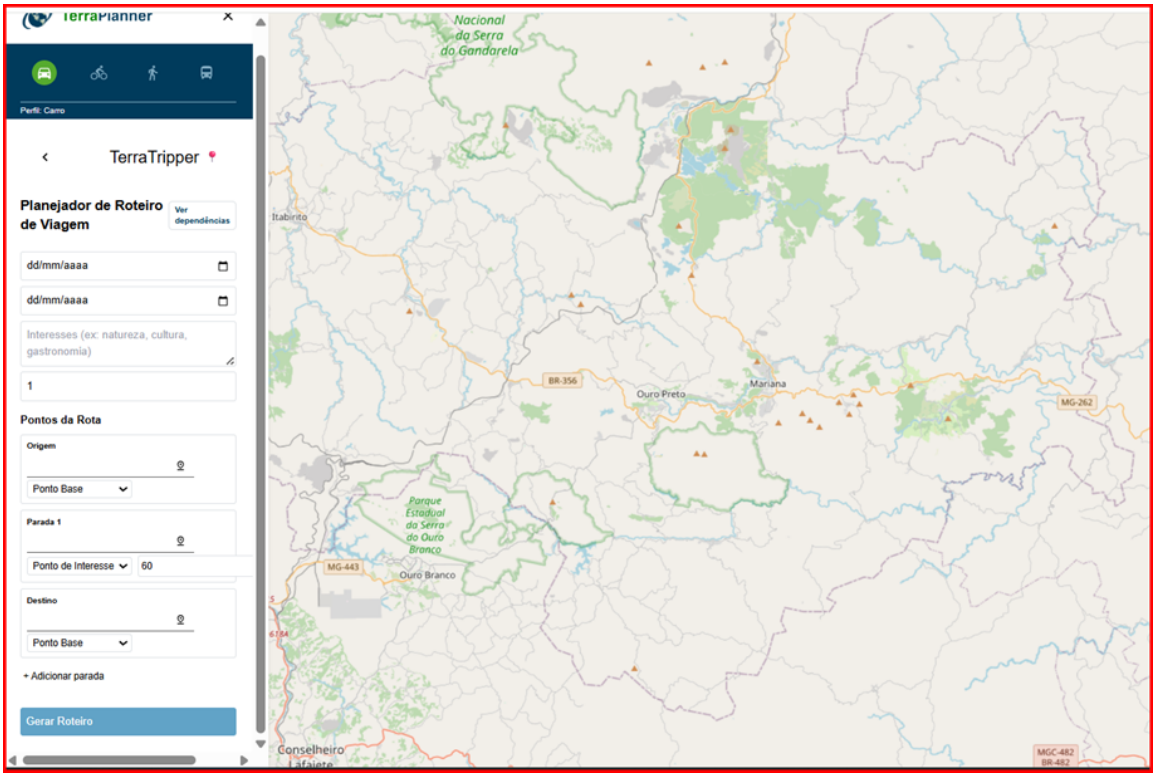

Figura 3.11 – Configuração inicial do TerraTripper (datas, interesses e parâmetros iniciais).

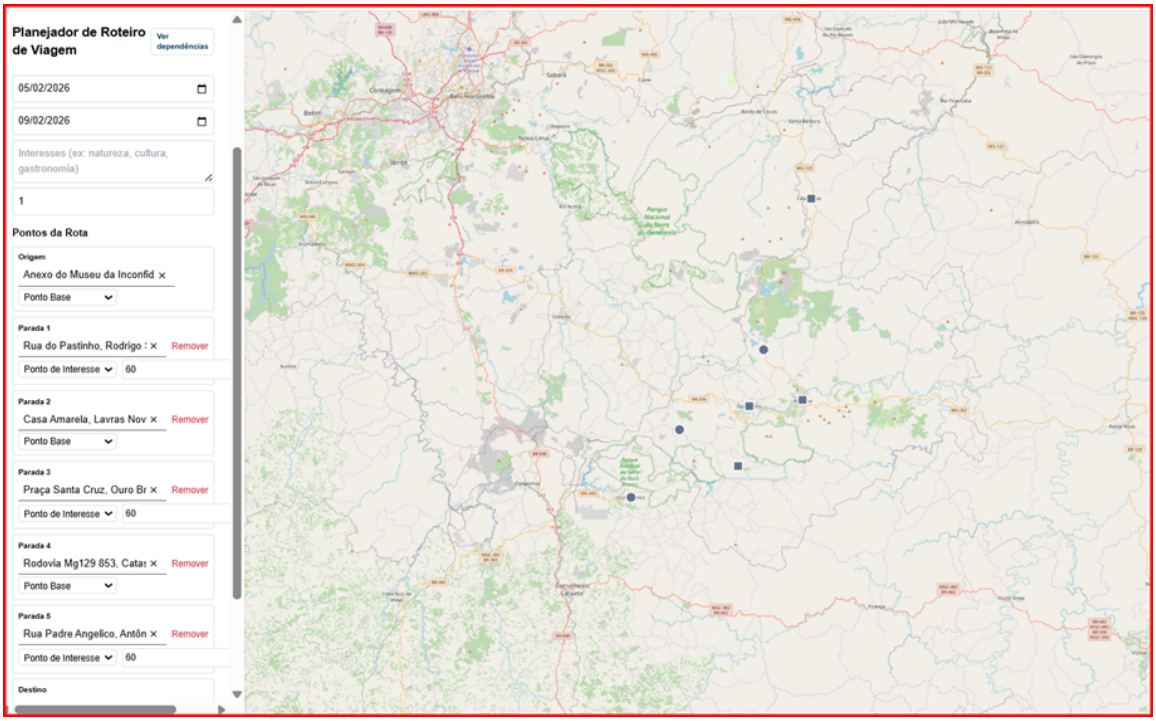

Figura 3.12 – Seleção de bases e pontos de interesse no mapa e na lista de paradas (à esquerda).

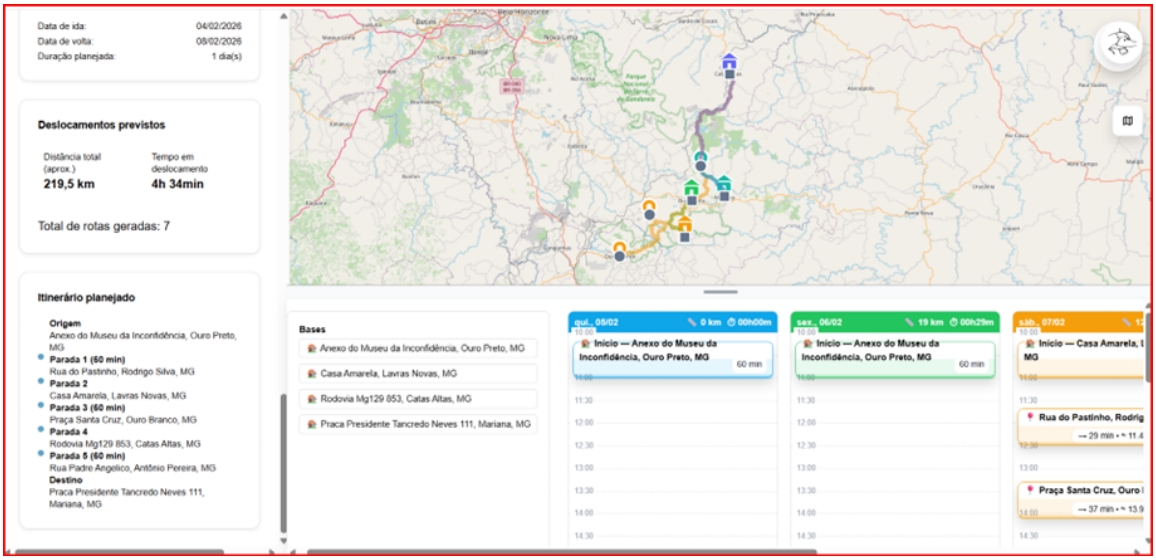

Figura 3.13 – Quadro de planejamento (calendário) com distribuição do itinerário por dia.

Figura 3.14 – Ajustes manuais no calendário por arraste de cartões (drag-and-drop) e edição de duração.

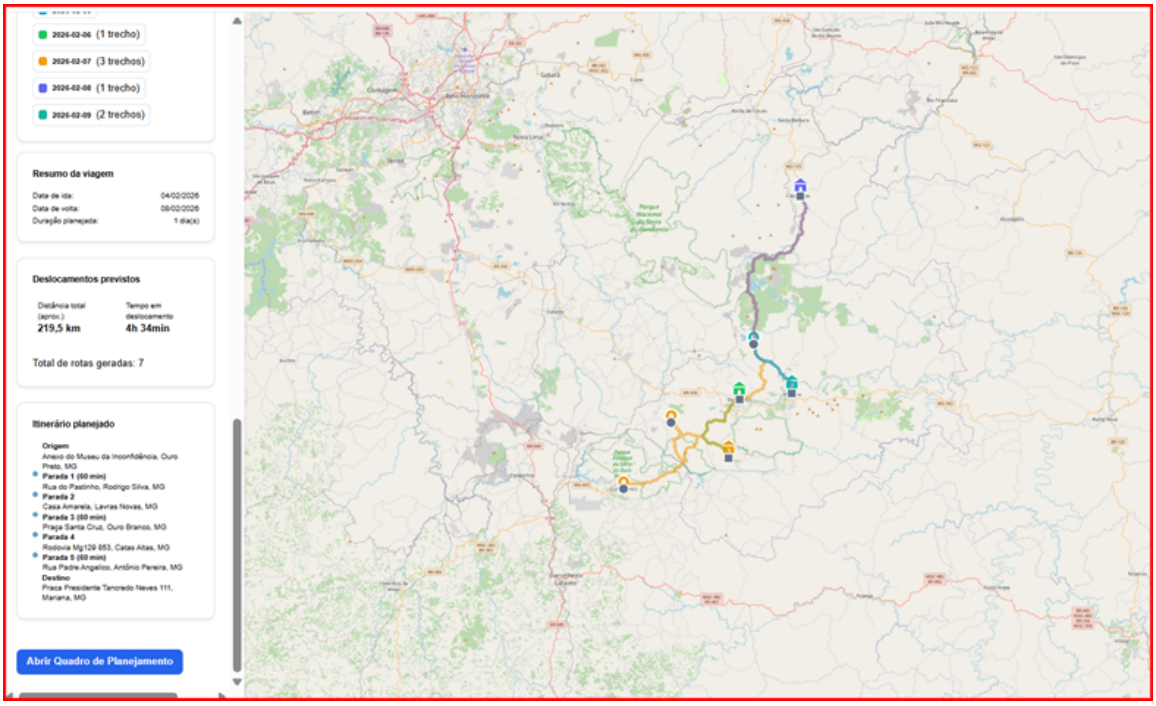

Figura 3.15 – Visualização de indicadores (tempo e distância) globais do roteiro e dos deslocamentos diários .

4 Resultados

Este capítulo apresenta os resultados produzidos ao longo da execução deste trabalho e os relaciona aos objetivos específicos definidos na Seção 1.2.2. Os artefatos técnicos que sustentam estes resultados (documentação de contratos, endpoints, guias e materiais complementares) encontram-se consolidados nos anexos/apêndices, de modo a facilitar o reuso e a verificação dos resultados. Além disso, os materiais complementares provenientes do TCC 1, incluindo o DER e o benchmarking, foram incorporados como anexos para dar suporte à rastreabilidade e à avaliação da solução proposta.

4.1 Entrega De Um MVP Funcional

Após a revisão da literatura sobre Geotecnologias, SDSS e abordagens para o desenvolvimento de arquiteturas de software extensíveis, visando desenvolver um SDSS extensível e avaliar sua viabilidade técnica, selecionou-se inicialmente uma plataforma de inteligência geográfica cuja arquitetura era fechada, isto é, sem recursos que permitissem a reutilização de seu código, a expansão de suas funcionalidades ou sua especialização para um domínio específico de aplicação. Em seguida, sua arquitetura foi modificada para torná-la aberta por meio de uma abordagem plugável. Foram adotadas medidas para reduzir acoplamento, preservar desempenho, controlar versões e estabelecer mecanismos de governança para extensões.

A arquitetura de plugins desenvolvida para o TerraPlanner constitui um produto mínimo viável (MVP, do inglês *Minimal Viable Product*) que permite: (i) upload e registro sob demanda de plugins React e dependências TypeScript via interface gráfica; (ii) listagem e carregamento dinâmico no frontend da plataforma hospedeira (TerraPlanner), com acesso controlado a elementos de interface e a serviços de inteligência espacial; e (iii) especialização por domínio por meio do plugin TerraTripper, que expande a interface e as funcionalidades da plataforma com serviços e componentes interativos voltados ao planejamento de viagens terrestres, incluindo recomputação automática de roteiros.

4.2 Validação Técnica

A validação do MVP foi conduzida por meio de uma prova de conceito, com foco em dois aspectos: (i) a consistência do comportamento do sistema sob recomputações sucessivas e (ii) a viabilidade da integração entre host e plugin no runtime. Em particular, avaliou-se a coerência dos roteiros produzidos a cada recomputação: verificou-se que alterações no calendário resultaram em mudanças determinísticas e reconstruíram trechos do roteiro na ordem correta, mitigando condições de corrida por meio de tokenização e da estratégia *last-write-wins*. Para

complementar a validação, os artefatos de rastreabilidade e avaliação oriundos do TCC 1 foram anexados: o DER (Anexo D) e o benchmarking (Anexo E), permitindo inspecionar requisitos, decisões e evidências adicionais de desempenho/viabilidade.

4.2.1 Análise arquitetural: vantagens, limitações e comparação

A arquitetura proposta favorece a personalização por domínio ao separar um núcleo estável (host) de extensões (plugins) que evoluem de forma independente. Entre as vantagens, destacam-se: (i) redução de acoplamento ao núcleo; (ii) menor custo de manutenção frente a múltiplos domínios; (iii) reuso de capacidades do host via contratos explícitos; e (iv) possibilidade de governança técnica (injeção controlada de dependências, compatibilidade e integração com serviços compartilhados). Como limitações, observam-se: (i) aumento da complexidade inicial (definição de contratos e pontos de extensão); (ii) necessidade de documentação e versionamento cuidadoso dos contratos; e (iii) riscos de segurança caso plugins sejam obtidos de fontes não confiáveis, o que demanda políticas de validação, controle de origem e auditoria.

Em comparação com a alternativa de customização por modificação direta do núcleo (por exemplo, *forks* e ramificações específicas por domínio), a abordagem plugável tende a reduzir divergências e retrabalho, pois preserva um núcleo comum e desloca a variabilidade para módulos externos. Em contraste com extensões restritas apenas ao backend, a abordagem proposta permite adaptar também a interface e o fluxo de decisão do usuário final, mantendo integração com os serviços base do host.

4.2.2 Experimento exploratório: desenvolvimento de plugin com apoio de IA

Foi conduzido um experimento exploratório para verificar se um agente de IA, munido da documentação e dos contratos expostos pelo host, seria capaz de implementar um plugin funcional. O procedimento consistiu em fornecer ao agente: (i) a descrição dos contratos disponíveis (APIs, hooks e componentes), (ii) um objetivo de plugin mínimo (renderizar uma interface, criar/selecionar pontos no mapa e acionar um serviço do host) e (iii) restrições de integração (uso exclusivo das dependências expostas pelo host). Como resultado, o plugin foi implementado e executado no runtime do TerraPlanner, consumindo os contratos previstos e respeitando as restrições definidas. Embora não substitua uma avaliação com múltiplos desenvolvedores humanos, o experimento reforça a adequação da arquitetura e da documentação para apoiar equipes de TI na criação de extensões.

4.3 Documentação Técnica

A arquitetura de plugins desenvolvida foi documentada, contemplando o detalhamento de endpoints, contratos, exemplos de uso e a especificação do objeto `PluginDependencies`. A documentação também registra decisões arquiteturais e trade-offs, visando orientar evoluções futuras e reduzir o custo de entrada para desenvolvedores que desejem produzir extensões sobre o host. Como material complementar, os anexos incluem o benchmarking (Anexo D) e o DER (Anexo E), os quais apoiam a rastreabilidade do desenvolvimento e a verificação dos resultados reportados neste capítulo.

5 Discussão e Trabalhos Futuros

A abordagem plugável demonstrou viabilidade para o desenvolvimento de SDSS na WEB, com baixo acoplamento e reuso de componentes e de serviços. Os resultados produzidos neste trabalho estão armazenados em repositorios Git do TerraLAB, aguardando o lançamento oficial da plataforma. Trabalhos futuros incluem: (i) cache e reuso de trechos de rota; (ii) suporte ao Problema de Roteamento de Veículos com Janelas de Tempo (VRPTW); (iii) uso critérios multiobjetivo no algoritmo de roteamento; (iv) avaliação com usuários por meio da Escala de Usabilidade do Sistema (SUS); (v) desenvolvimento de catálogo público de plugin; e (vi) produtação do MVP por meio de testes alpha e beta nos próximos meses.

6 Conclusão

Este trabalho desenvolveu e avaliou a viabilidade técnica de um framework extensível para o desenvolvimento de ferramentas de apoio à tomada de decisão geoespacial, capacitando a plataforma TerraPlanner a carregar plugins em seu frontend garantindo o compartilhamento de dependências, de serviços de inteligência geográfica e componentes de interface gráfica. O plugin TerraTripper evidenciou de maneira efetiva viabilidade técnica da abordagem proposta ao permitir o planejamento interativo de viagens com recomputação incremental e automática de rotas. Os resultados indicam que a modularidade e a injeção controlada de dependências favorecem evolução e reuso de software, com potencial para o desenvolvimento de novas aplicações para diversos outros domínios de aplicação.

Referências

ARMSTRONG, M. P. Requirements for the development of gis-based group decision-support systems. *Journal of the American Society for Information Science*, v. 45, n. 9, p. 669–677, 1994.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Chichester: Wiley, 1996.

DENSHAM, P. J.; GOODCHILD, M. F. Spatial decision support systems: A research agenda. *Journal of Environmental Sciences (China) English Ed*, Scanning Microscopy International, p. 707–716, 1989.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1994.

GeoNode Project. *GeoNode Documentation*. [S.l.], 2024. Disponível em: <https://docs.geonode.org/>.

GOODCHILD, M. F. Citizens as sensors: the world of volunteered geography. *GeoJournal*, v. 69, n. 4, p. 211–221, 2007.

GORRY, G. A.; MORTON, M. S. S. A framework for management information systems. *Sloan Management Review*, v. 13, n. 1, p. 55–70, 1971.

JANKOWSKI, P.; NYERGES, T. L.; SMITH, A.; MOORE, T.; HORVATH, E. Spatial group choice: a sdss tool for collaborative spatial decisionmaking. *International journal of geographical information science*, Taylor & Francis, v. 11, n. 6, p. 577–602, 1997.

LONGLEY, P. A.; GOODCHILD, M. F.; MAGUIRE, D. J.; RHIND, D. W. *Geographic Information Systems and Science*. 4. ed. Hoboken: Wiley, 2015.

MALCZEWSKI, J. *GIS and Multicriteria Decision Analysis*. New York: John Wiley & Sons, 1999.

MALCZEWSKI, J. Gis-based multicriteria decision analysis: a survey of the literature. *International Journal of Geographical Information Science*, 2006.

MCAFFER, J.; LEMIEUX, J. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Boston: Addison-Wesley, 2005.

Open Geospatial Consortium. *OpenGIS® Web Map Server Implementation Specification*. [S.l.], 2006. Disponível em: <https://portal.ogc.org/files/?artifact_id=14416>.

Open Geospatial Consortium. *OpenGIS Web Feature Service 2.0 Interface Standard*. 2010. Disponível em: <https://docs.ogc.org/is/09-025r2/09-025r2.html>.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, v. 15, n. 12, p. 1053–1058, 1972.

PENG, Z.; TSOU, M. *Internet GIS: Distributed Geographic Information Services for the Internet and Wireless Networks*. Hoboken, NJ: Wiley, 2003.

QGIS Development Team. *QGIS Documentation – Python Plugins*. [S.l.], 2026. Disponível em: <https://docs.qgis.org/>.

React Team. *React Documentation*. 2026. Acesso em: 17 fev. 2026. Disponível em: <https://react.dev/>.

SOMMERVILLE, I. Software engineering (ed.). *America: Pearson Education Inc*, 2011.

SPRAGUE, R. H. A framework for the development of decision support systems. *MIS Quarterly*, v. 4, n. 4, p. 1–26, 1980.

SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. 2. ed. Boston: Addison-Wesley, 2002.

Anexos

ANEXO A – Documentação da Arquitetura Plugável do TerraPlanner

Documentação da Arquitetura Plugável do TerraPlanner

1. Introdução

O TerraPlanner é um Sistema de Apoio à Decisão Espacial (SADE) que evoluiu de uma arquitetura monolítica para um sistema dinâmico e extensível. Este documento detalha a arquitetura plugável implementada, que permite a adição de novas funcionalidades em tempo de execução sem a necessidade de modificar o código-fonte principal. A solução foi projetada para ser robusta, flexível e de fácil manutenção, atendendo tanto às necessidades acadêmicas quanto às de desenvolvimento prático.

A refatoração arquitetural teve como objetivo principal desacoplar funcionalidades, permitindo que módulos independentes, chamados **plugins**, sejam desenvolvidos e integrados de forma isolada. Uma API de backend foi criada para gerenciar o ciclo de vida desses plugins, desde o upload até a sua disponibilização para o frontend.

2. Visão Geral da Arquitetura

A nova arquitetura do TerraPlanner é composta por três componentes principais: o **Frontend TerraPlanner**, a **Plugin API (Backend)** e os **Plugins Externos**. A comunicação entre eles é orquestrada para garantir isolamento e segurança, utilizando um sistema de contratos bem definidos para a interação.

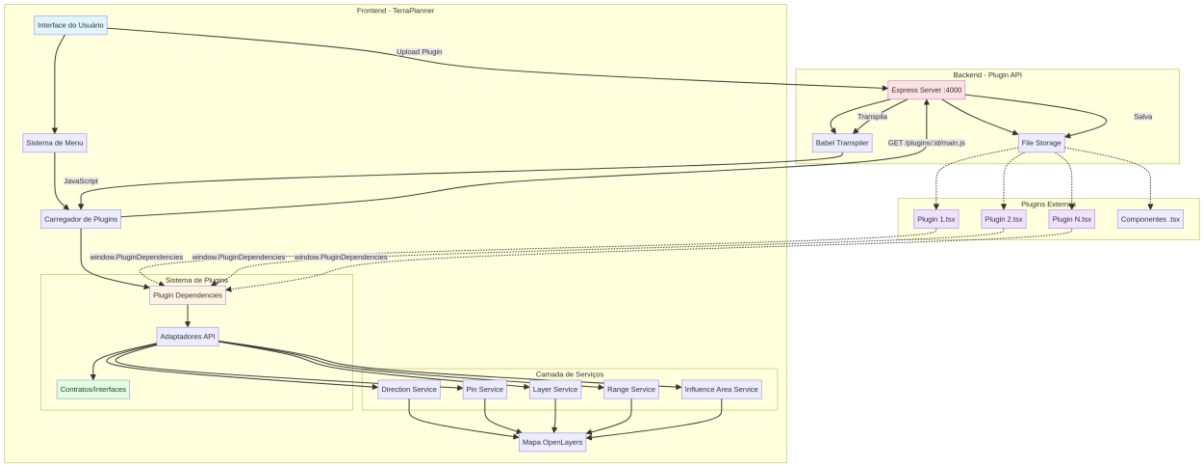

Figura 1: Diagrama de Arquitetura Geral do Sistema TerraPlanner.

2.1. Componentes do Sistema

A tabela abaixo resume os principais componentes e suas responsabilidades dentro da arquitetura.

Componente	Tecnologia	Responsabilidades Principais
Frontend (TerraPlanner)	React 19, TypeScript, Vite, OpenLayers	- Renderização da interface principal e do mapa. - Gerenciamento do ciclo de vida dos plugins (carregamento e exibição). - Injeção de dependências para os plugins via window.PluginDependencies. - Definição dos contratos de API.
Plugin API (Backend)	Node.js, Express, Babel	- Receber upload de arquivos de plugin (.tsx). - Armazenar os plugins em um diretório dedicado. - Transpilar o código TSX para JavaScript em tempo real (Just-in-Time). - Servir os plugins transpilados para o frontend.
Plugins Externos	React, TypeScript	- Implementar funcionalidades específicas (ex: análise de rotas, visualização de dados). - Interagir com o sistema principal através dos contratos expostos. - Ser carregado dinamicamente pelo frontend.

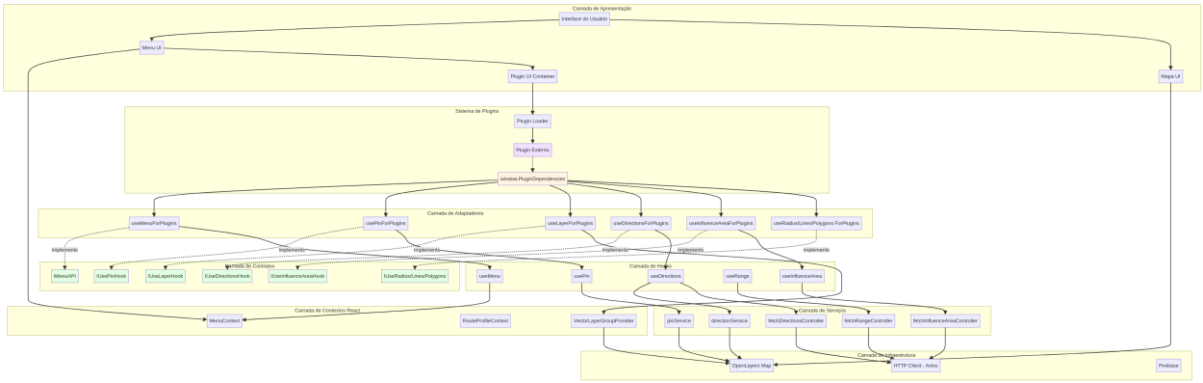

Figura 2: Visão detalhada dos componentes e suas interações.

3. Fluxo de Dados e Ciclo de Vida do Plugin

O ciclo de vida de um plugin abrange desde sua criação e upload até seu carregamento e execução no navegador do usuário. Esse processo foi desenhado para ser o mais transparente possível para o desenvolvedor do plugin.

3.1. Fluxo de Upload de um Novo Plugin

Um desenvolvedor pode adicionar um novo plugin diretamente pela interface do TerraPlanner. O processo envolve o envio de um arquivo .tsx principal e, opcionalmente, outros componentes auxiliares.

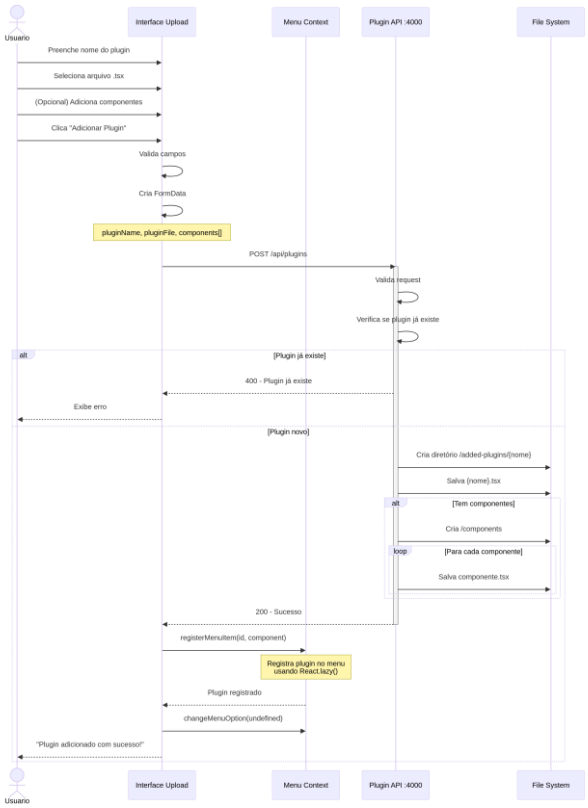

Figura 3: Sequência de eventos durante o upload de um novo plugin.

O fluxo ocorre da seguinte forma:

1. O usuário preenche um formulário no frontend com o nome do plugin e os arquivos .tsx.
2. O frontend envia esses dados via POST /api/plugins para a Plugin API.
3. A API armazena os arquivos no diretório added-plugins.
4. Após o sucesso, o frontend registra o novo plugin no sistema de menu, tornando-o disponível para o usuário.

3.2. Fluxo de Carregamento e Transpilação

Quando um usuário decide ativar um plugin, o frontend o carrega dinamicamente. A Plugin API é responsável por preparar o código para execução no navegador.

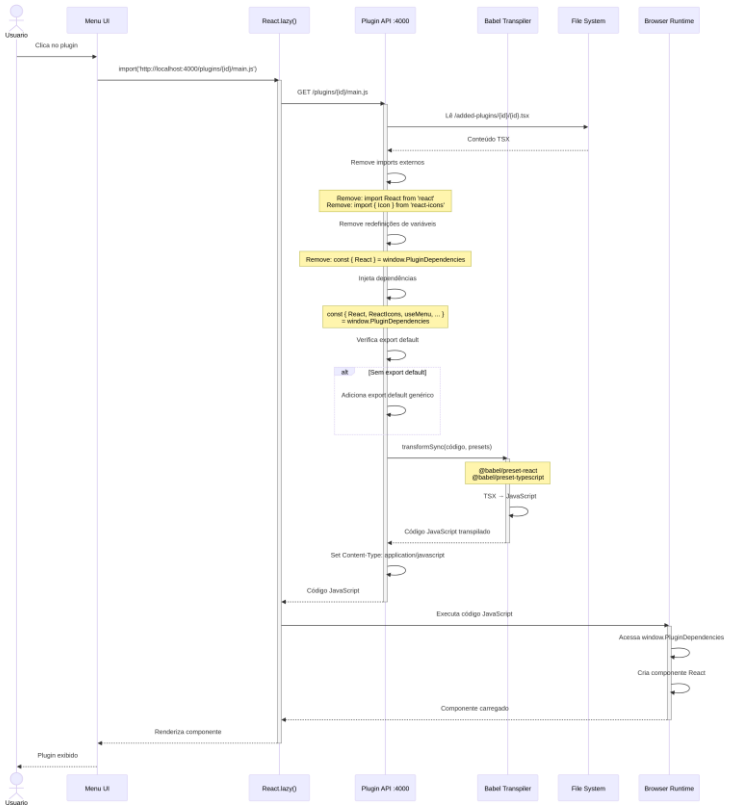

Figura 4: Processo de carregamento e transpilação JIT de um plugin.

As etapas são:

5 O frontend utiliza React.lazy() para solicitar o plugin via import('http://localhost:4000/plugins/{id}/main.js').

6 A Plugin API lê o arquivo .tsx correspondente.

7 **Injeção de Dependências**: A API modifica o código em memória para remover imports de bibliotecas globais (como react) e injeta o objeto window.PluginDependencies.

8 **Transpilação**: O código modificado é transpilado de TSX para JavaScript usando Babel.

9 O JavaScript resultante é enviado ao frontend, que o executa e renderiza o componente do plugin.

4. Arquitetura de Contratos e Injeção de Dependências

Para garantir o desacoplamento e a estabilidade, os plugins não acessam as funcionalidades internas do TerraPlanner diretamente. Em vez disso, eles dependem de um conjunto de **contratos** (interfaces TypeScript) que definem as APIs disponíveis. Essa abordagem segue o **Princípio da Inversão de Dependência**.

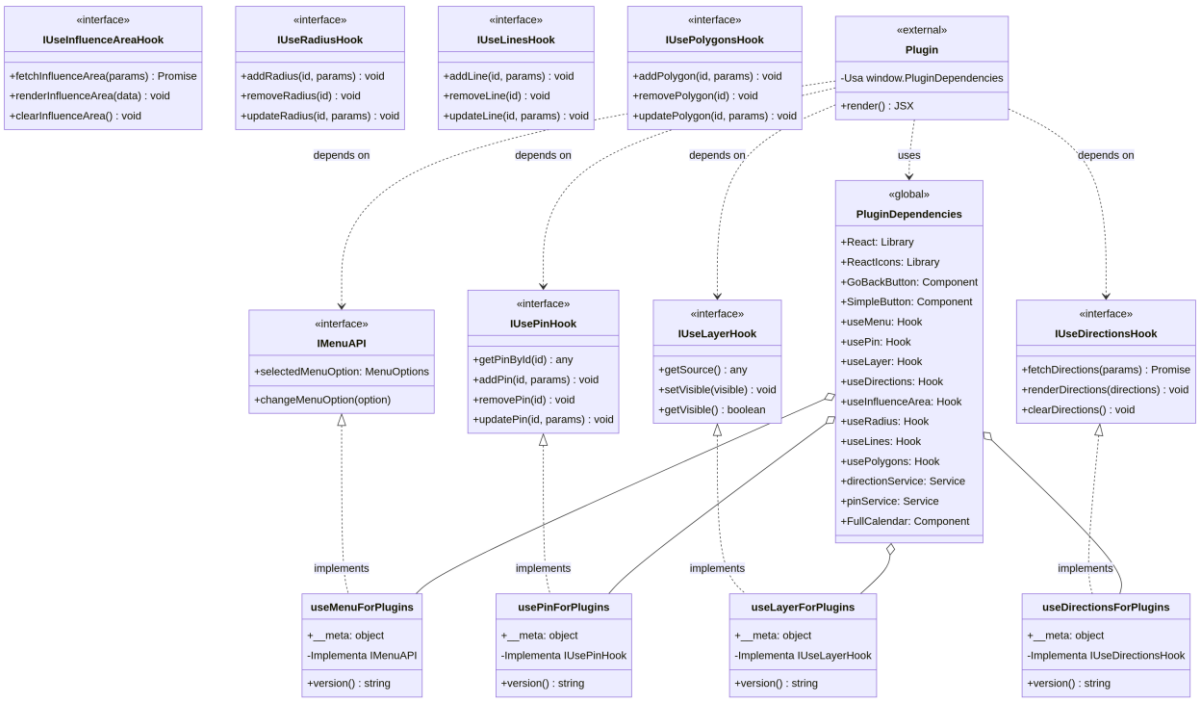

Figura 5: Relação entre plugins, contratos e o objeto de dependências.

4.1. O Objeto window.PluginDependencies

No momento da inicialização, o TerraPlanner popula um objeto global, window.PluginDependencies, com todas as funções, hooks e componentes que os plugins podem utilizar. Isso inclui:

- **Bibliotecas Essenciais**: React e ReactIcons.
- **Componentes de UI**: GoBackButton, SimpleButton para manter a consistência visual.
- **Hooks de Funcionalidades**: useMenu, usePin, useLayer, useDirections, entre outros.
- **Serviços e Controladores**: Acesso a funcionalidades mais complexas, como cálculo de rotas e áreas de influência.

4.2. Contratos de API

Os contratos estão localizados em /src/plugins/contracts e definem a "superfície de contato" da API para os plugins. Alguns dos principais contratos são:

Contrato	Descrição	Exemplo de Uso
IUseMenuHook	Controla o estado do menu lateral, permitindo que o plugin mude a visão atual.	const { changeMenuOption } = useMenu();

Contrato	Descrição	Exemplo de Uso
IUsePinHook	Permite adicionar, remover e atualizar marcadores (pins) no mapa.	addPin('my-pin', { coordinates: [-43, -20] });
IUseLayerHook	Fornece acesso a camadas (layers) do mapa para manipulação de visibilidade e dados.	const myLayer = useLayer('my-layer'); myLayer.setVisible(true);
IUseDirectionsHook	Utilizado para calcular e renderizar rotas entre pontos no mapa.	fetchDirections({ profile: 'driving-car', coordinates });
IUseInfluenceAreaHook	Permite a criação e visualização de isócronas (áreas de influência).	fetchInfluenceArea({ point, range_type: 'time', contours_minutes: [5, 10] });

5. Guia para Desenvolvedores de Plugins

Criar um novo plugin para o TerraPlanner é um processo simples, projetado para focar na funcionalidade em si, e não na complexidade de integração.

5.1. Estrutura de um Plugin

Um plugin é, em sua essência, um componente React escrito em TSX. Ele deve ter uma exportação padrão (export default).

Exemplo de Template de Plugin (meu-plugin.tsx):

```
// As dependências são injetadas globalmente, não precisam de import.

const { React, useMenu, GoBackButton, SimpleButton, ReactIcons } =
window.PluginDependencies;
const { BiSmile } = ReactIcons.Bi;

export default function MeuPlugin() {
  const [count, setCount] = React.useState(0);
  const { changeMenuOption } = useMenu();

  return (
    <main className="w-full p-4">
      <GoBackButton title="Meu Plugin" icon={BiSmile} onGoBack={() =>
changeMenuOption(undefined)} />

      <h2 className="text-xl font-bold mb-4">Plugin de Exemplo</h2>
      <p>Contador: {count}</p>

      <SimpleButton onClick={() => setCount(count + 1)}>
```

```
          Incrementar
        </SimpleButton>
      </main>
    );
}
```

5.2. Utilizando Dependências

Para usar uma funcionalidade do TerraPlanner, basta desestruturá-la do objeto window.PluginDependencies. Por exemplo, para interagir com o mapa, um desenvolvedor pode usar o hook usePin.

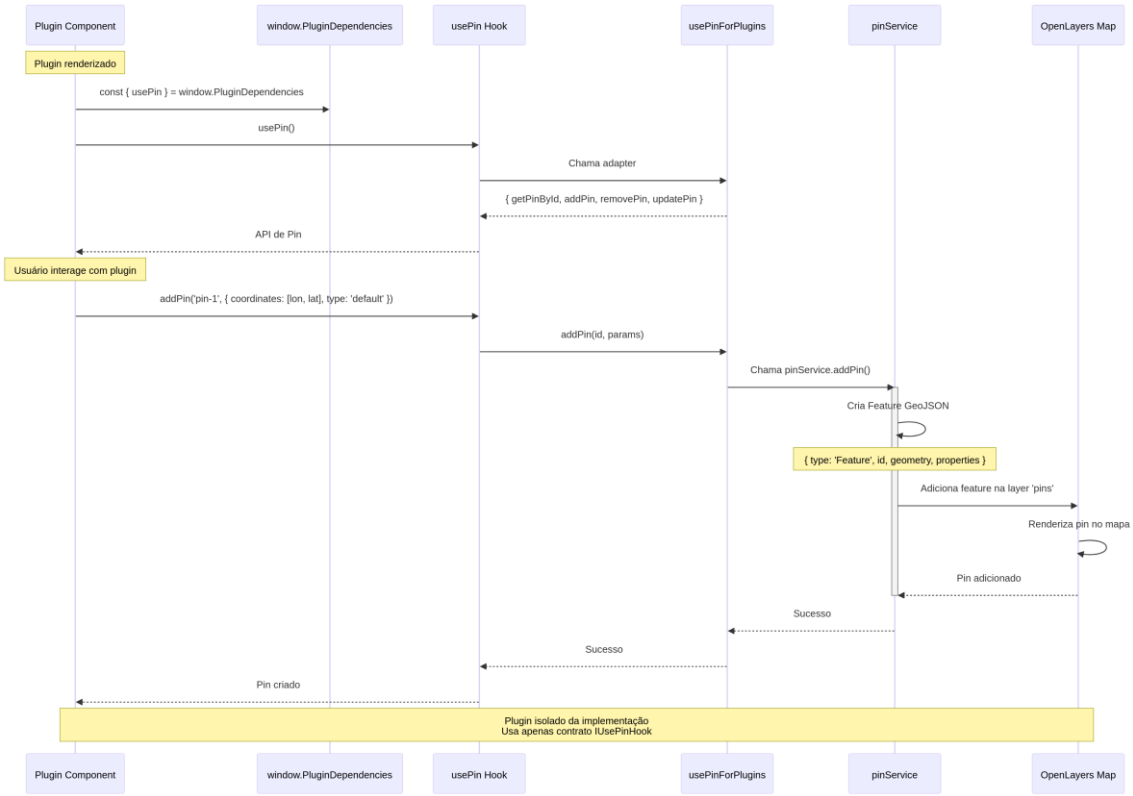

Figura 6: Exemplo de como um plugin utiliza o usePin para adicionar um marcador no mapa.

5.3. Componentes Auxiliares

Se um plugin for complexo, ele pode ser dividido em múltiplos arquivos. Crie uma pasta components junto ao arquivo principal do plugin. Os componentes auxiliares podem ser importados com caminhos relativos.

Estrutura de Arquivos:

```
meu-plugin/
```

```
├── meu-plugin.tsx
└── components/
    └── BotaoCustomizado.tsx
```

Importação no Plugin Principal:

```
// meu-plugin.tsx

const BotaoCustomizado = React.lazy(() =>
import('./components/BotaoCustomizado.js'));

// Note que a extensão é .js, pois a API se encarrega da transpilação.
```

6. Conclusão

A transição do TerraPlanner para uma arquitetura plugável representa um avanço significativo em termos de manutenibilidade, escalabilidade e flexibilidade. Ao definir contratos claros e fornecer um mecanismo robusto para injeção de dependências e carregamento dinâmico, o sistema permite que a comunidade acadêmica e desenvolvedores externos contribuam com novas funcionalidades de forma segura e isolada.

Esta abordagem não apenas resolve os desafios de um sistema monolítico, mas também abre portas para um ecossistema de inovações em torno da plataforma TerraPlanner, consolidando seu papel como uma ferramenta poderosa de apoio à decisão espacial.

ANEXO B – Guia Rápido: Como Criar um Plugin para o TerraPlanner

Guia Rápido: Como Criar um Plugin para o TerraPlanner

Este guia prático destina-se a desenvolvedores que desejam estender as funcionalidades do TerraPlanner através da criação de plugins personalizados.

Pré-requisitos

- Conhecimento básico de React e TypeScript.
- Um ambiente de desenvolvimento com Node.js para testar componentes (opcional).

Passo 1: Crie o Arquivo Principal do Plugin

O coração do seu plugin é um único arquivo .tsx que exporta um componente React como padrão (export default).

Crie um arquivo chamado meu-primeiro-plugin.tsx:

```
// meu-primeiro-plugin.tsx

// 1. Desestruture as dependências que você precisa do objeto global.
const { React, GoBackButton, useMenu, ReactIcons } =
window.PluginDependencies;
const { BiRocket } = ReactIcons.Bi;

// 2. Crie seu componente React.
export default function MeuPrimeiroPlugin() {
  // Use os hooks e componentes do TerraPlanner como se fossem nativos.
  const { changeMenuOption } = useMenu();

  return (
    <main className="w-full p-4 bg-gray-50 h-full">
      {/* O GoBackButton gerencia o retorno ao menu principal */}
      <GoBackButton
        title="Meu Primeiro Plugin"
        icon={BiRocket}
        onGoBack={() => changeMenuOption(undefined)}
      />

      <div className="mt-4">
        <h2 className="text-xl font-bold">🚀 Plugin Carregado com
Sucesso!</h2>
        <p className="mt-2 text-gray-700">
```

```
          Este componente está sendo renderizado dinamicamente pelo
TerraPlanner.
        </p>
      </div>
    </main>
  );
}
```

Passo 2: Adicione Interatividade com o Mapa

Vamos usar o hook usePin para adicionar um marcador no mapa quando o plugin é carregado.

```
// meu-plugin-com-mapa.tsx

const { React, GoBackButton, useMenu, usePin, ReactIcons } =
window.PluginDependencies;
const { BiMapPin } = ReactIcons.Bi;

export default function PluginComMapa() {
  const { changeMenuOption } = useMenu();
  const { addPin, removePin } = usePin();

  const PIN_ID = 'meu-plugin-pin';

  // Adiciona um pin quando o componente é montado.
  React.useEffect(() => {
    addPin(PIN_ID, {
      coordinates: [-43.93449, -19.91668], // Coordenadas de Belo
Horizonte
      type: 'destination',
    });

    // Remove o pin quando o componente é desmontado (ao fechar o
plugin).
    return () => {
      removePin(PIN_ID);
    };
  }, [addPin, removePin]); // Adicione as dependências ao array

  return (
    <main className="w-full p-4">
      <GoBackButton title="Plugin com Mapa" icon={BiMapPin} onGoBack={()
=> changeMenuOption(undefined)} />
      <div className="mt-4">
```

```
        <h2 className="text-xl font-bold">📍 Marcador Adicionado!</h2>
        <p className="mt-2 text-gray-700">
          Verifique o mapa. Um marcador foi adicionado em Belo Horizonte.
        </p>
      </div>
    </main>
  );
}
```

Passo 3: Organize seu Código com Componentes

Para plugins mais complexos, você pode dividir a lógica em componentes auxiliares.

1 **Crie a estrutura de pastas:**

```
meu-plugin-complexo/
├── meu-plugin-complexo.tsx
└── components/
    └── PainelDeControle.tsx
```

2 **Crie o componente auxiliar (PainelDeControle.tsx):**

```tsx
// components/PainelDeControle.tsx
const { React, SimpleButton } = window.PluginDependencies;

interface PainelProps {
  onAction: () => void;
  label: string;
}

export default function PainelDeControle({ onAction, label }:
PainelProps) {
  return (
    <div className="p-4 border rounded-md mt-4">
      <p className="mb-2">Este é um painel de controle interno.</p>
      <SimpleButton onClick={onAction}>{label}</SimpleButton>
    </div>
  );
}
```

3 **Use o componente no plugin principal (meu-plugin-complexo.tsx):**

```tsx
// meu-plugin-complexo.tsx

const { React, GoBackButton, useMenu } = window.PluginDependencies;

// Carregue o componente dinamicamente.
// Use a extensão .js no import!
const PainelDeControle = React.lazy(() =>
import('./components/PainelDeControle.js'));

export default function MeuPluginComplexo() {
  const { changeMenuOption } = useMenu();

  function handleAction() {
    alert('Ação do painel executada!');
  }

  return (
    <main className="w-full p-4">
      <GoBackButton title="Plugin Complexo" onGoBack={() =>
changeMenuOption(undefined)} />

      <React.Suspense fallback={<div>Carregando painel...</div>}>
        <PainelDeControle onAction={handleAction} label="Executar Ação"
/>
      </React.Suspense>
    </main>
  );

}
```

Passo 4: Faça o Upload do Plugin

4. No menu do TerraPlanner, encontre a opção "Adicionar Plugin" (ou similar).
5. **Nome do Plugin**: Dê um nome único, como Meu Primeiro Plugin.
6. **Arquivo Principal**: Faça o upload do seu arquivo .tsx principal (ex: meu-primeiro-plugin.tsx).
7. **Componentes Adicionais**: Se você criou componentes auxiliares, faça o upload de todos os arquivos da pasta components.
8. Clique em **"Adicionar Plugin"**.

Seu plugin aparecerá no menu, pronto para ser usado!

APIs Disponíveis em window.PluginDependencies

Consulte a documentação técnica principal para a lista completa de hooks e componentes disponíveis. Os mais comuns são:

- React, ReactIcons
- GoBackButton, SimpleButton
- useMenu: Para controle de navegação.
- usePin: Para manipular marcadores.
- useLayer: Para interagir com camadas do mapa.
- useDirections: Para calcular rotas.
- useInfluenceArea: Para calcular isócronas.
- useRadius, useLines, usePolygons: Para desenhar no mapa.

Com estes passos, você está pronto para estender o TerraPlanner com suas próprias funcionalidades geoespaciais.

ANEXO C – Referência Técnica de APIs para Plugins TerraPlanner

Referência Técnica de APIs para Plugins TerraPlanner

Este documento serve como referência rápida para desenvolvedores de plugins, listando todas as APIs, hooks e componentes disponíveis através do objeto window.PluginDependencies.

Bibliotecas e Componentes de UI

React

```
const { React } = window.PluginDependencies;
```

Biblioteca React completa, incluindo hooks como useState, useEffect, useCallback, useMemo, etc.

ReactIcons

```
const { ReactIcons } = window.PluginDependencies;
```

```
const { BiHome, BiUser } = ReactIcons.Bi;
```

```
const { FiSettings } = ReactIcons.Fi;
```

Biblioteca completa de ícones do React Icons. Principais coleções:

- Bi - Bootstrap Icons
- Fi - Feather Icons
- Ai - Ant Design Icons
- Fa - Font Awesome
- Md - Material Design Icons

GoBackButton

```
const { GoBackButton } = window.PluginDependencies;
```

```
<GoBackButton
  title="Título do Plugin"
  icon={BiIcon}
  onGoBack={() => changeMenuOption(undefined)}
```

```
/>
```

Componente padrão para cabeçalho de plugins com botão de voltar.

Props:

- title: string - Título exibido
- icon: IconType - Ícone do React Icons
- onGoBack: () => void - Callback ao clicar em voltar

SimpleButton

```
const { SimpleButton } = window.PluginDependencies;
```

```
<SimpleButton
  onClick={handleClick}
  className="bg-blue-500 hover:bg-blue-700 text-white"
>
  Clique aqui
</SimpleButton>
```

Botão estilizado padrão do sistema.

Props:

- onClick: () => void - Callback de clique
- className?: string - Classes CSS adicionais
- children: ReactNode - Conteúdo do botão

Hooks de Contexto e Navegação

useMenu

```
const { useMenu } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const { selectedMenuOption, changeMenuOption } = useMenu();

  // Voltar ao menu principal
  changeMenuOption(undefined);

  // Navegar para outra opção
```

```
  changeMenuOption('outro-plugin');
```

```
}
```

Retorno:

```
interface IMenuAPI {
```

```
  selectedMenuOption: MenuOptions | undefined;
  changeMenuOption(menuOption: MenuOptions | undefined): void;
```

```
}
```

Hooks de Mapa e Geometria

usePin

```
const { usePin } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const { getPinById, addPin, removePin, updatePin } = usePin();

  // Adicionar pin
  addPin('meu-pin', {
    coordinates: [-43.93449, -19.91668],
    type: 'default' // 'default' | 'destination' | 'tr_poi' | 'tr_base'
  });

  // Atualizar pin
  updatePin('meu-pin', {
    coordinates: [-43.94, -19.92],
    type: 'destination'
  });

  // Remover pin
  removePin('meu-pin');

  // Obter pin
  const pin = getPinById('meu-pin');
```

```
}
```

Interface:

```
interface IUsePinHookResult {
```

```
  getPinById(id: string): any;
  addPin(id: string, opts: AddPinParams): void;
  removePin(id: string): void;
  updatePin(id: string, opts: UpdatePinParams): void;
}

interface AddPinParams {
  coordinates: number[]; // [longitude, latitude]
  type?: 'default' | 'destination' | 'tr_poi' | 'tr_base';
}

interface UpdatePinParams {
  coordinates?: number[];
  type?: 'default' | 'destination' | 'tr_poi' | 'tr_base';
```

```
}
```

useLayer

```
const { useLayer } = window.PluginDependencies;
```

```

function MeuPlugin() {
  const myLayer = useLayer('nome-da-layer');

  if (myLayer) {
    // Mostrar/ocultar layer
    myLayer.setVisible(true);

    // Verificar visibilidade
    const isVisible = myLayer.getVisible();

    // Acessar fonte de dados
    const source = myLayer.getSource();
  }
```

```
}
```

Interface:

```
interface TerraLayerLike {
```

```
  getSource?(): any;
  setVisible?(visible: boolean): void;
  getVisible?(): boolean;
  [key: string]: any;
}
```

```
type IUseLayerHook = (name: string) => TerraLayerLike | undefined;
```

useRadius

```
const { useRadius } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const { addRadius, removeRadius, updateRadius, listRadii, clearSource }
= useRadius();

  // Adicionar círculo
  addRadius('meu-circulo', {
    center: [-43.93449, -19.91668],
    radiusMeters: 5000, // 5km
    style: {
      strokeColor: '#FF0000',
      fillColor: 'rgba(255, 0, 0, 0.2)',
      strokeWidth: 2
    }
  });

  // Atualizar raio
  updateRadius('meu-circulo', {
    radiusMeters: 10000
  });

  // Remover círculo
  removeRadius('meu-circulo');

  // Listar todos os círculos
  const circulos = listRadii();

  // Limpar todos
  clearSource();
```

```
}
```

useLines

```
const { useLines } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const { addLine, removeLine, updateLine, listLines, clearSource } =
useLines();

  // Adicionar linha
  addLine('minha-linha', {
    coordinates: [
      [-43.93449, -19.91668],
      [-43.94, -19.92],
      [-43.95, -19.93]
    ],
    style: {
      strokeColor: '#0000FF',
      strokeWidth: 3
    }
  });

  // Remover linha
  removeLine('minha-linha');
```

```
}
```

usePolygons

```
const { usePolygons } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const { addPolygon, removePolygon, updatePolygon, listPolygons,
clearSource } = usePolygons();

  // Adicionar polígono
  addPolygon('meu-poligono', {
    coordinates: [[
      [-43.93, -19.91],
      [-43.94, -19.91],
      [-43.94, -19.92],
      [-43.93, -19.92],
      [-43.93, -19.91] // Fechar o polígono
    ]],
    style: {
      strokeColor: '#00FF00',
      fillColor: 'rgba(0, 255, 0, 0.3)',
```

```
      strokeWidth: 2
    }
  });

  // Remover polígono
  removePolygon('meu-poligono');
```

```
}
```

Hooks de Análise Espacial

useDirections

```
const { useDirections } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const {
    fetchDirections,
    renderDirections,
    clearDirections,
    listDirections
  } = useDirections();

  // Calcular rota
  const directions = await fetchDirections({
    profile: 'driving-car', // 'driving-car' | 'foot-walking' | 'cycling-
regular'
    coordinates: [
      [-43.93449, -19.91668],
      [-43.94, -19.92]
    ]
  });

  // Renderizar no mapa
  renderDirections([directions]);

  // Limpar rotas
  clearDirections();
```

```
}
```

Profiles disponíveis:

- 'driving-car' - Carro
- 'foot-walking' - Caminhando
- 'cycling-regular' - Bicicleta

useRange

```
const { useRange } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const {
    fetchRange,
    renderRanges,
    listRanges,
    clearSource
  } = useRange();

  // Calcular alcance (isócrona)
  const range = await fetchRange({
    profile: 'driving-car',
    locations: [[-43.93449, -19.91668]],
    range_type: 'time', // 'time' | 'distance'
    range: [300, 600, 900] // 5min, 10min, 15min em segundos
  });

  // Renderizar no mapa
  renderRanges([range]);

  // Limpar
  clearSource();
```

```
}
```

useInfluenceArea

```
const { useInfluenceArea } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const {
    fetchInfluenceArea,
    renderInfluenceArea,
    clearInfluenceArea,
    listInfluenceAreas
  } = useInfluenceArea();
```

```
  // Calcular área de influência
  const area = await fetchInfluenceArea({
    profile: 'driving-car',
    locations: [[-43.93449, -19.91668]],
    range_type: 'time',
    contours_minutes: [5, 10, 15]
  });

  // Renderizar
  renderInfluenceArea(area);

  // Limpar
  clearInfluenceArea();
```

```
}
```

useRouteProfile

```
const { useRouteProfile } = window.PluginDependencies;
```

```
function MeuPlugin() {
  const { profile, changeProfile } = useRouteProfile();

  // Obter perfil atual
  console.log(profile); // 'driving-car'

  // Mudar perfil
  changeProfile('foot-walking');
```

```
}
```

Serviços Imperativos

directionService

```
const { directionService } = window.PluginDependencies;
```

```
// Serviço para manipulação imperativa de rotas
directionService.clear();
```

```
directionService.addDirection(directionData);
```

pinService

```
const { pinService } = window.PluginDependencies;
```

```

// Serviço imperativo de pins
await pinService.clear();
await pinService.addPins([/* features GeoJSON */]);
```

```
await pinService.setPinStyle('pin-id', { colorHex: '#FF0000' });
```

Controllers HTTP

fetchDirectionsController

```
const { fetchDirectionsController } = window.PluginDependencies;
```

```

const result = await fetchDirectionsController.fetch({
  profile: 'driving-car',
  coordinates: [[lon1, lat1], [lon2, lat2]]
```

```
});
```

fetchRangeController

```
const { fetchRangeController } = window.PluginDependencies;
```

```

const result = await fetchRangeController.fetch({
  profile: 'driving-car',
  locations: [[lon, lat]],
  range_type: 'time',
  range: [300, 600]
```

```
});
```

fetchInfluenceAreaController

```
const { fetchInfluenceAreaController } = window.PluginDependencies;
```

```

const result = await fetchInfluenceAreaController.fetch({
  profile: 'driving-car',
  locations: [[lon, lat]],
  range_type: 'time',
  contours_minutes: [5, 10, 15]
```

```
});
```

FullCalendar (Calendário)

Componentes e Plugins

```
const {
```

```
  FullCalendar,
  fcTimeGrid,
  fcInteraction,
  fcScrollGrid,
  FullCalendarDraggable,
  fcLocalePtBr
} = window.PluginDependencies;

function MeuPlugin() {
  return (
    <FullCalendar
      plugins={[fcTimeGrid, fcInteraction, fcScrollGrid]}
      initialView="timeGridWeek"
      locale={fcLocalePtBr}
      events={[
        { title: 'Evento', start: '2025-11-28T10:00:00' }
      ]}
    />
  );
```

```
}
```

Exemplo Completo de Plugin

```
// plugin-analise-rota.tsx
```

```
const {
  React,
  GoBackButton,
  SimpleButton,
  useMenu,
  useDirections,
  usePin,
  ReactIcons
} = window.PluginDependencies;

const { BiRoute } = ReactIcons.Bi;

export default function AnaliseRota() {
  const { changeMenuOption } = useMenu();
  const { fetchDirections, renderDirections, clearDirections } =
useDirections();
  const { addPin, removePin } = usePin();
  const [loading, setLoading] = React.useState(false);

  async function calcularRota() {
    setLoading(true);

    // Adicionar pins de origem e destino
    addPin('origem', {
      coordinates: [-43.93449, -19.91668],
      type: 'default'
    });

    addPin('destino', {
      coordinates: [-43.94, -19.92],
      type: 'destination'
    });

    // Calcular rota
    const rota = await fetchDirections({
      profile: 'driving-car',
      coordinates: [
        [-43.93449, -19.91668],
        [-43.94, -19.92]
      ]
    });
```

```
    renderDirections([rota]);
    setLoading(false);
  }

  function limpar() {
    clearDirections();
    removePin('origem');
    removePin('destino');
  }

  return (
    <main className="w-full p-4">
      <GoBackButton
        title="Análise de Rota"
        icon={BiRoute}
        onGoBack={() => changeMenuOption(undefined)}
      />

      <div className="mt-4 space-y-4">
        <SimpleButton
          onClick={calcularRota}
          className="bg-blue-500 hover:bg-blue-700 text-white"
          disabled={loading}
        >
          {loading ? 'Calculando...' : 'Calcular Rota'}
        </SimpleButton>

        <SimpleButton
          onClick={limpar}
          className="bg-red-500 hover:bg-red-700 text-white"
        >
          Limpar
        </SimpleButton>
      </div>
    </main>
  );
```

```
}
```

Notas Importantes

1. **Não use import**: Todas as dependências vêm de window.PluginDependencies.
2. **Extensão .js em imports relativos**: Ao importar componentes auxiliares, use .js mesmo que o arquivo seja .tsx.
3. **Coordenadas**: Sempre no formato [longitude, latitude].

4 **IDs únicos**: Use IDs únicos para pins, linhas, polígonos, etc.

5 **Cleanup**: Sempre limpe recursos (pins, rotas) quando o componente for desmontado usando useEffect com retorno de função.

ANEXO D – Benchmarking

Análise das Ferramentas para apoio à tomada de Decisão Espacial

Levantamento de requisitos

Para identificar os requisitos e funcionalidades essenciais do TerraTripper, foi conduzido um benchmarking de ferramentas similares no campo do planejamento de viagens. Os requisitos escolhidos para a análise foram selecionados a partir da literatura proposta, baseando-se nas funcionalidades básicas que um SDSS deve possuir e no foco da pesquisa. São eles:

- Conexão com a Internet: uma conexão estável com a internet pode ser necessária para atualizações em tempo real e sincronização de dados.
- Mapas offline: capacidade de baixar mapas para uso offline, útil em áreas com pouca conectividade.
- Serviço de login: permitir aos usuários criar contas pessoais para acesso personalizado.
- Sincronia de dados em tempo real: atualização em tempo real de informações, como pontos de interesse e dados de trânsito.
- Serviços de mapeamento e navegação: integração de serviços de mapeamento e navegação para orientação precisa.
- Planejamento com múltiplos apontadores: capacidade de planejar itinerários com várias paradas.
- Estimativas de tempo: fornecer estimativas de tempo de viagem para melhor planejamento.
- Criação de itinerários: permitir aos usuários criar e personalizar itinerários de viagem.
- Compartilhamento de itinerários: capacidade de compartilhar itinerários com outros usuários.
- Informações sobre trânsito: fornecer dados sobre tráfego em tempo real para evitar congestionamentos.
- Compartilhamento de dados: permitir o compartilhamento de informações relevantes entre usuários ou dispositivos.

Benchmarking

A tabela apresentada a seguir reúne ferramentas populares no contexto do planejamento de viagens e indica se elas atendem ou não aos requisitos elencados para o desenvolvimento de um SDSS. As ferramentas

selecionadas foram as seguintes:

- Google Maps: sistema multiplataforma que oferece serviços de mapeamento, navegação e planejamento de viagens. Com recursos como mapas offline, estimativas de tempo, criação de itinerários e informações de trânsito em tempo real, permite explorar, planejar e navegar por diferentes destinos. Também possibilita o compartilhamento de itinerários e dados.
- TripAdvisor: plataforma multiplataforma voltada à consulta de avaliações e informações turísticas. Embora não se concentre em mapeamento ou planejamento detalhado, permite aos usuários compartilhar experiências e avaliações, contribuindo para decisões mais informadas durante o planejamento de viagens.
- Wickloc: plataforma voltada à organização de experiências e informações de viagem, com ênfase em colaboração e compartilhamento entre usuários.
- RoadTrippers: ferramenta voltada ao planejamento de viagens rodoviárias, com foco em rotas, múltiplas paradas, estimativas de percurso e organização de itinerários. Destaca-se pelo apoio ao planejamento visual de viagens em estrada.
- TripIt: sistema que organiza itinerários de viagem ao coletar e estruturar automaticamente informações de reservas de voos, hotéis e atividades. Embora não ofereça recursos avançados de mapeamento, é útil para centralizar detalhes essenciais da viagem.

	FERRAMENTAS:	Google Maps	TripAdvisor	Wickloc	RoadTrippers	TripIt
REQUISITOS	Multiplataforma?	√	√	√	√	√
	Requer conexão com internet?	√	√	√	√	√
	Sincronia de dados em tempo real (Mapas e informações sobre pontos de interesse)?	√	X	√	X	X
	Mapas offline?	√	X	√	√	X
	Serviço de login?	√	√	√	√	√
	Serviços de mapeamento e navegação?	√	X	√	√	X
	Planejamento com múltiplos apontadores?	√	X	√	√	X
	Estimativas de tempo?	√	X	√	√	X
	Criação de itinerários?	√	X	√	√	√
	Compartilhamento de itinerários?	√	X	√	√	√
	Informações sobre trânsito?	√	X	√	√	X
	Compartilhamento de dados?	√	√	√	√	√

Tabela - Benchmarking das principais ferramentas e seus requisitos no contexto de planejamento de viagens.

Fonte: Produzida pelo próprio autor.

ANEXO E – DER - Documento de especificação de requisitos - TerraTripper

Concepção do DER (Documento de Especificação de Requisitos)

Com base nos requisitos e funcionalidades identificados por meio da análise das ferramentas similares, irá ser desenvolvida uma abordagem para a criação do sistema TerraTripper.

O DER tem como objetivo descrever as funcionalidades e serviços a serem desenvolvidos na aplicação TerraTripper . Serão apresentados os objetivos e comportamentos esperados de cada serviço e qual problema do cliente ele resolve. Serão indicados também as limitações do produto e o backlog do projeto.

Descrição do Produto

Escopo do Produto

Atualmente, existem várias ferramentas de apoio à tomada de decisão voltadas para o planejamento de viagens, que proporcionam aos usuários informações como custos, rotas, atrações, entre outras, que colaboram diretamente no planejamento de roteiros e viagens. entre elas, estão os Portais Geocolaborativos (GCP). Esses portais têm como finalidade permitir que os usuários compartilhem e colaborem na criação, visualização e edição de informações geográficas, como mapas, rotas, pontos de interesse, avaliações e comentários, de forma gratuita. Porém, muitas das ferramentas GCP existentes apresentam uma barreira significativa em relação à complexidade técnica e à falta de interfaces de usuário intuitivas (SIEBER, 2006).

O que é o produto

Um Sistema de Apoio à Decisão Espacial (SADE), no formato de um Portal Geocolaborativo (GCP) voltado para o planeamento de viagens.

Nome do produto e de seus componentes principais

O sistema terá dois componentes : • uma aplicação Web, denominada TerraTripper, que irá exibir e tratar dados geoespaciais, a fim de fornecer informações e roteiros para uma determinada viagem que o usuario queira fazer. Esses dados serão baseados em indicadores, préviamente especificados pelo usuario. • Serviço Web denominado TerraPlanner, que irá fornecer as informações geográficas e geoespaciais solicitadas, baseando-se nos parâmetros recebidos.

Missão do Produto

O sistema deverá ser capaz de apoiar um usuário nas tomadas de decisão no planejamento de uma viagem, no contexto de visitar cachoeiras(indicador geográfico escolhido para fins da pesquisa) durante o percurso. Esse apoio será baseado em análises de dados geoespaciais fornecidos pelo usuário, permitindo que rotas sejam traçadas e exibindo estatísticas relevantes para o itinerário, fornecendo ao usuário uma visão clara e detalhada sobre seu planejamento.

Limites do Produto

A aplicação não toma decisões, ela irá apenas exibir as informações relacionadas à uma viagem seguindo especificações do usuario, para auxiliar na tomada de decisão.

Beneficios do Produto

O produto tem como principal objetivo maximizar a experiência do usuario e sua tomada de decisão em relação à roteiros de viagens voltadas para o turismo, exibindo roteiros compartilhados por outros usuarios e permitindo a personalização de apontadores especificos para um melhor planejamento da viagem.

Serviços que serão oferecidos pelo produto

Requisitos Funcionais

- **Serviço de autenticação :**

  - Login e Logoff

- **Serviço de Gerenciamento de Usuários:**

  - Cadastro, remoção, edição e exclusão de usuários na plataforma web.

  - Gerenciamento de perfis de usuário e preferências.

- **Serviço de Informações sobre Cachoeiras (Indicador geográfico escolhido):**

  - Criação, compartilhamento, edição, exclusão e exibição de informações detalhadas sobre cachoeiras (apontadores iniciais). As informações poderão conter textos e imagens.

  - Integração com API externa para obter dados de cachoeiras.

  -Avaliações de imagens e descrições das cachoeiras.

- **Serviço de Planejamento de Rotas com Cachoeiras (Indicador geográfico escolhido):**

  - Criação, compartilhamento, edição e exclusão de rotas para visitar múltiplas cachoei-

- Geração de estimativas de tempo de viagem entre as cachoeiras.
- Personalização de rotas com base nas preferências do usuário.

- **Serviço de Personalização de Indicadores :**
  - Possibilidade de usuários criarem seus próprios indicadores personalizados.
  - Adição de descrições e informações personalizadas aos indicadores.

- **Serviço de Armazenamento e Sincronização:**
  - Armazenamento seguro de rotas, indicadores e dados do usuário.
  - Sincronização de rotas, indicadores e preferências entre dispositivos.

- **Serviço de Análise e Estatísticas:**
  - Coleta de dados de uso para análises e melhorias contínuas.
  - Análise de rotas populares, indicadores personalizados e preferências dos usuários.

Requisitos não-funcionais

- **Segurança e Privacidade:**
  - Implementação de medidas de segurança para proteger dados sensíveis.
  - Controles de privacidade para gerenciar compartilhamento de informações.

- **Desempenho:**
  - Resposta rápida para a exibição de informações e itinerários.
  - Processamento eficiente das rotas e cálculos de tempo de viagem.

- **Usabilidade e Experiência do Usuário:**
  - Interface de usuário intuitiva e fácil de usar.
  - Navegação clara e eficiente nas rotas e indicadores.

- **Disponibilidade:**
  - Garantia de disponibilidade da plataforma para os usuários.

- **Escalabilidade:**
  - Capacidade de lidar com um grande número de usuários e rotas.

- **Compatibilidade:**

    - Compatibilidade com diferentes navegadores web.

- **Confiabilidade:**

    - Minimização de falhas e erros no sistema.

    - Manutenção de dados e rotas consistentes.

- **Documentação:**

    - Documentação clara e abrangente para os desenvolvedores e usuários.

- **Performance da API:**

    - Resposta rápida e baixa latência ao se comunicar com a API externa.

- **Integração com API Externa:**

    - Integração confiável e segura com a API externa de customização de indicadores.

Generalização dos Atores

De acordo com os princípios da Engenharia de Requisitos e da Linguagem de Modelagem Unificada (UML), um 'Diagrama de Atores' é uma ferramenta utilizada para representar as interações entre diferentes atores, como usuários, sistemas externos ou componentes, e o sistema em desenvolvimento. Esse tipo de diagrama ilustra os papéis desempenhados por esses atores e ajuda a identificar como eles interagem com o sistema, contribuindo para uma compreensão mais clara dos requisitos do projeto.(SOMMERVILLE, 2011)

Identifica-se na Figura 1 as principais relações entre os atores que interagem com o sistema apartir de um diagrama de atores:

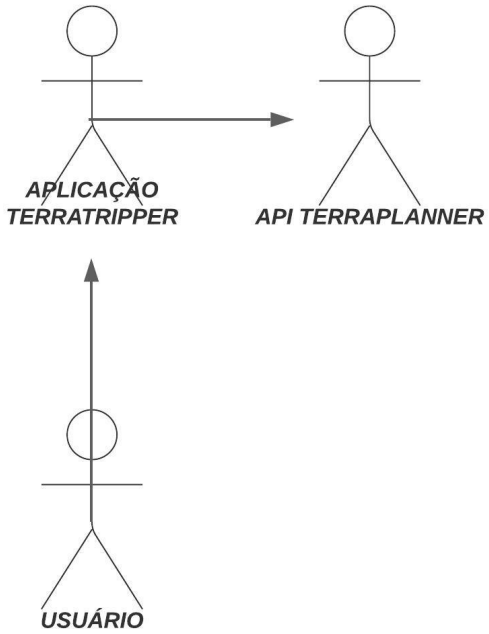

Figura – Diagrama de Atores.

Fonte: Produzida pelo próprio autor

Descrição dos Atores

Identifica-se na tabela os principais atores que interagem com o sistema.

Número	Ator	Descrição	Nível de Instrução	Proficiência na Aplicação
1	Usuário	O usuário é o principal ator. Ele interage com a aplicação TerraTripper para criar rotas, definir indicadores e realizar outras atividades relacionadas ao planejamento de viagens.	N/A	Alta
2	Aplicação TerraTripper	A aplicação TerraTripper é o sistema em que o usuário interage diretamente. Ela oferece a interface e as funcionalidades para criar rotas, definir indicadores,visualizar mapas e realizar outras tarefas de planejamento de viagens.	N/A	Alta
3	API TerraPlanner	A API TerraPlanner é um componente que opera nos bastidores. Ela recebe os dados e solicitações da aplicação TerraTripper e realiza o tratamento e processamento dos dados para gerar indicadores, analisar rotas, calcular estimativas de tempo, avaliar dados espaciais, entre outros	N/A	Alta

Tabela – Lista de Atores

Fonte: Produzida pelo próprio autor

Diagrama de contexto

Um Diagrama de Contexto é uma ferramenta usada para representar visualmente a interação entre um sistema e seus elementos externos, como atores externos, outros sistemas, pessoas ou entidades. Ele é uma representação de alto nível que ajuda a entender o contexto no qual o sistema está inserido, sem entrar em detalhes internos(SOMMERVILLE, 2011). A figura esenta o diagrama de contexto do sistema, demonstrando a iteração dos Atores com os serviços propostos.

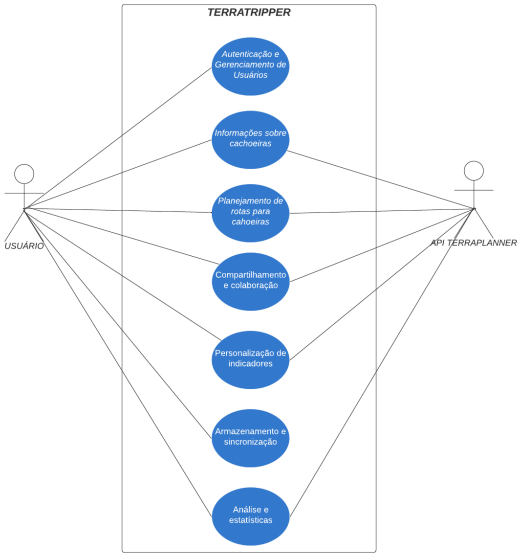

Figura – Diagrama de Atores.

Fonte: Produzida pelo próprio autor

Descrição dos Serviços

Identifica-se na tabela as principais funções que o produto desempenhará :

Número	Caso de Uso	Descrição
1	Autenticação e Gerenciamento de Usuários	O usuário é capaz de se cadastrar na aplicação, além de poder gerenciar o seu perfil
2	Informações sobre cachoeiras	O usuário tem acesso às informações disponíveis sobre as cachoeiras presentes na rota traçada.(As informações são providas pela API TERRAPLANNER)
3	Planejamento de rotas para cachoeiras	O sistema será capaz de traçar uma rota entrer dois ou mais pontos, baseando-se nas preferências do usuário e levando em consideração o indicador cachoeira.
4	Compartilhamento e colaboração	Os usuários serão capazes de compartilhar suas rotas préviamente planejadas.
5	Personalização de indicadores	O usuário poderá personalizar os indicadores geográficos para um planejamento de viagens mais preciso.
6	Armazenamento e sincronização	O sistema armazena informações de usuários como rotas, preferências de perfil, entre outros. Esses dados serão sincronizados e apresentados em tempo real
7	Análise e Estatísticas	O usuário tem acesso à estatisticas e análises sobre rotas, no intuito de refinar o planejamento de viagem.

Tabela – Lista de Serviços do TerraTripper.

Fonte: Produzida pelo próprio autor

Storyboards

Descreve-se nessa seção os requisitos relacionados à interface com o usuário e as interfaces com demais componentes externos, sejam estes de software ou hardware.

A tabela descreve as interfaces consideradas essenciais para que o software funcione de forma intuitíva.

Número	Interface	Atores	Casos de Uso	Descrição
1	Janela de login	Usuário	Para ter acesso ao software, o usuário deve possuir autenticação	Janela de autenticação do software.
2	Janela Principal	Usuário	Após a autenticação, o usuário será levado à essa página.	Janela principal do sistema, contendo um mapa geográfico, o menu principal (sidebar) e as opções do software serão apresentadas.
3	Menu Principal (Sidebar)	Usuário	O menu apresenta ao usuário as funcionalidades principais da aplicação.	Nessa janela, o sistema exibe todas as funcionalidades que o usuário poderá acessar, são elas : Gestão de perfil, Rotas, e indicadores.
4	Janela de gestão de perfil	Usuário	acessar o perfil e/ou fazer LogOff.	Janela que possibilita o usuário editar as informações de seu perfil e fazer LogOff.
5	Janela de Rotas	Usuário	Planejar um itinerário.	Janela que permite ao usuario escolher um ponto de origem, um ponto de chegada, o indicador usado para calculo de rotas e o botão para calcular rotas.
6	Janela de indicadores	Usuário	Selecionar indicador(es) para o planejamento de viagem	O sistema irá exibir os indicadores disponíveis (no caso será somente cachoeiras), e , futuramente, permitirá a criação de novos indicadores.

Tabela – Lista de interfaces de usuário - TerraTripper

Fonte: Produzida pelo próprio autor

Backlog

O backlog do projeto apresentado pela figura foi concebido à partir da análise dos requisitos funcionais que devem ser oferecidos pelo produto. Visando a arquitetura de microserviços, houve um remanejamento das funcionalidades.

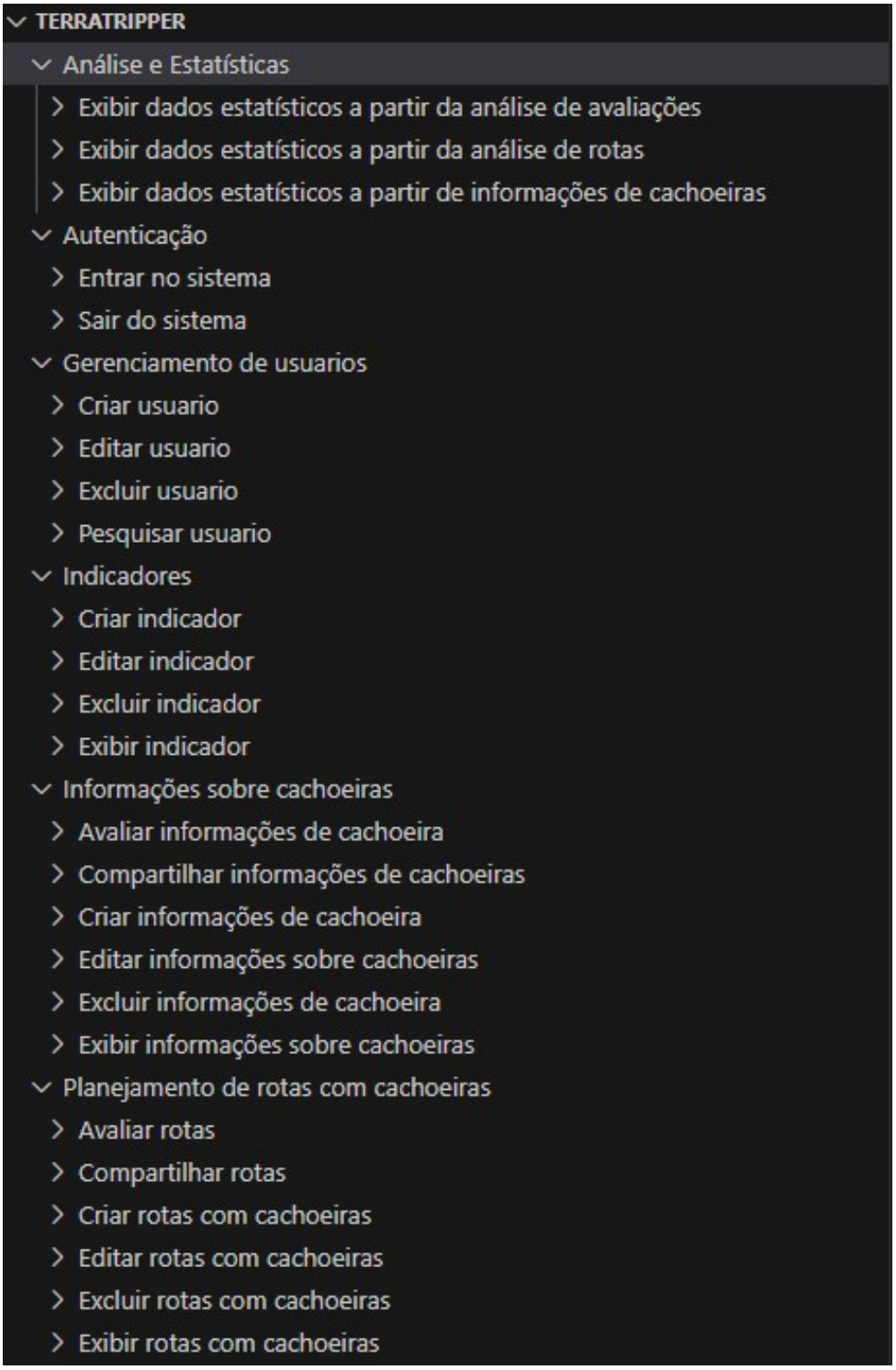

Figura – Backlog - TerraTripper

Fonte: Produzida pelo próprio autor

MVP

A partir do backlog, foi definido o MVP (Mínimo produto viável), levando-se em consideração o funcionamento básico do sistema e o tempo da pesquisa. A figura apresenta as funcionalidades que serão desenvolvidas no MVP. Essas funcionalidades estão grifadas em amarelo.

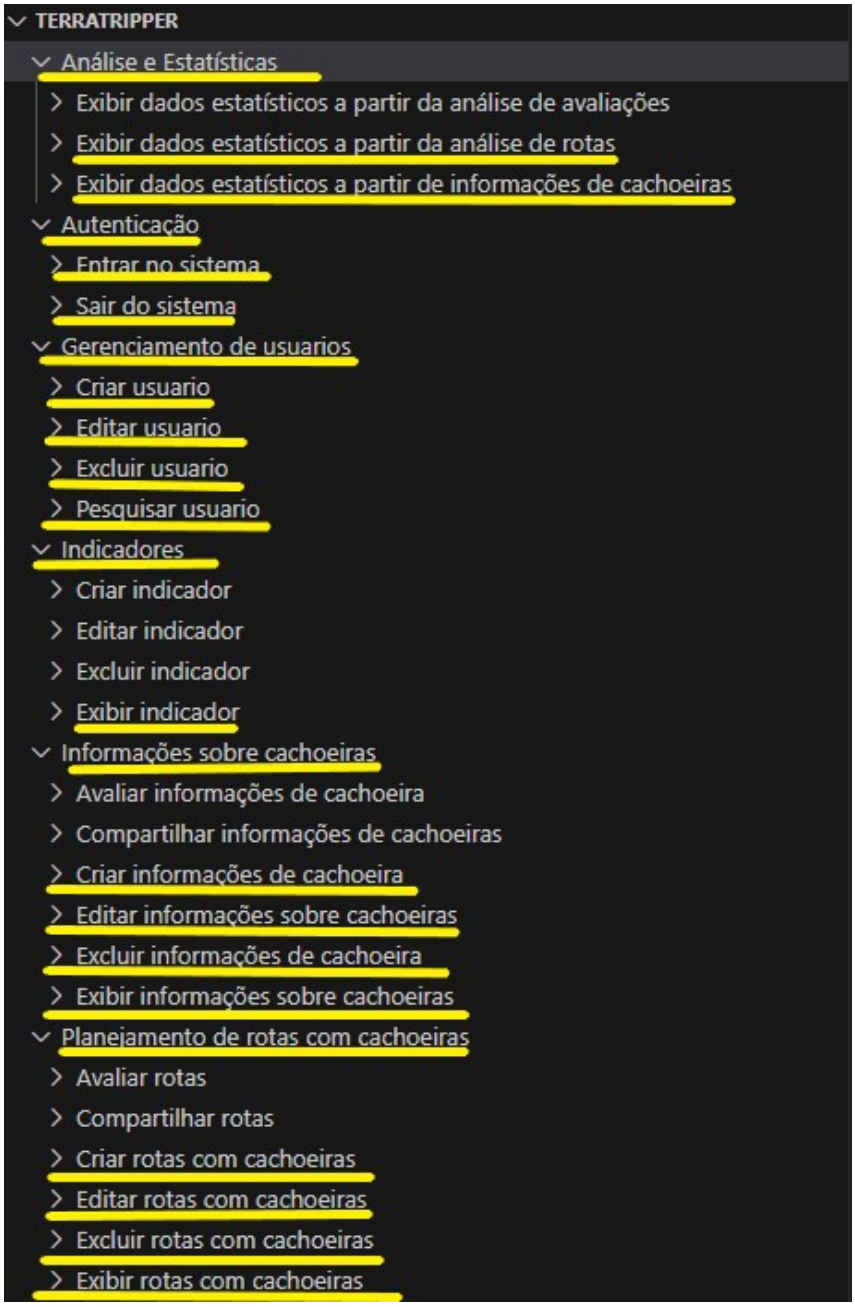

Figura – MVP - TerraTripper

Fonte: Produzida pelo próprio autor